17. Memory Corruption Exploits
ENEE 757 | CMSC 818V

Prof. Tudor Dumitraș
Assistant Professor, ECE
University of Maryland, College Park

http://ter.ps/757
https://www.facebook.com/SDSAtUMD

Today’s Lecture
• Where we’ve been
  – Authentication and access control
  – Network security
• Where we’re going today
  – Starting new module: distributed infrastructures supporting cybercrime
  – Memory corruption exploits
• Where we’re going next
  – Worms and infection spreading
Recall: Correctness versus Security

• System **correctness**: system satisfies specification
  – For reasonable input, get reasonable output

• System **security**: system properties preserved in face of attack
  – For unreasonable input, output not completely disastrous

• Main difference: **intelligent adversary trying to subvert system and to evade defensive techniques**

Memory Exploits

• **Buffer** is a data storage area inside computer memory (stack or heap)
  – Intended to hold pre-defined amount of data
  – If executable code is supplied as “data”, victim’s machine may be fooled into executing it
    • Code will disclose information or give attacker control over machine
  – Many attacks do not involve executing “data” (e.g. Heartbleed)

• Attack can exploit any memory operation
  – Pointer assignment, format strings, memory allocation and de-allocation, function pointers, calls to library routines via offset tables ...
Stack Buffers

• Suppose Web server contains this function

```c
void func(char *str) {
    char buf[126];
    strcpy(buf,str);
}
```

• When this function is invoked, a new frame is pushed onto the stack

What If Buffer Is Overstuffed?

• Memory pointed to by str is copied onto stack...

```c
void func(char *str) {
    char buf[126];
    strcpy(buf,str);
}
```

• If a string longer than 126 bytes is copied into buffer, it will overwrite adjacent stack locations
Executing Attack Code

- Suppose buffer contains attacker-created string
  - For example, str points to a string received from the network as the URL

  In the overflow, a pointer back into the buffer appears in the location where the program expects to find return address.

  Attacker puts actual assembly instructions into his input string, e.g., binary code of `execve("/bin/sh")`

- When function exits, code in the buffer will be executed, giving attacker a shell
  - Root shell if the victim program is setuid root

Stack Corruption: General View

```c
int bar (int val1) {
    int val2;
    foo (a_function_pointer);
}

int foo (void (*funcp)()) {
    char* ptr = point_to_an_array;
    char buf[128];
    gets (buf);
    strncpy(ptr, buf, 8);
    (*funcp)();
}
```

- Most popular target
- Stack Corrup.on: General View
- String grows
- Stack grows
Attack #1: Return Address

① Change the return address to point to the attack code. After the function returns, control is transferred to the attack code.
② ... or return-to-libc: use existing instructions in the code segment such as system(), exec(), etc. as the attack code.

Basic Stack Code Injection

[Aleph One – Smashing the Stack for Fun and Profit]

- Executable attack code is stored on stack, inside the buffer containing attacker’s string
  - Stack memory is supposed to contain only data, but...

- For the basic stack-smashing attack, overflow portion of the buffer must contain correct address of attack code in the RET position
  - The value in the RET position must point to the beginning of attack assembly code in the buffer
  - If you return outside the valid address space, the application will crash with segmentation violation
  - Attacker must correctly guess in which stack position his buffer will be when the function is called
**Fundamental Causes for Basic Stack Smashing Exploits**

- C strings are nul-terminated, rather than specifying the bound
  - Programmer must check the range manually
  - Standard C library functions are all unsafe
    - `strcpy(char *dest, const char *src)`
    - `strcat(char *dest, const char *src)`
    - `gets(char *)`
    - `scanf(const char *format, ...)`
    - `printf(const char *format, ...)`

- Stacks grow down and arrays grow up

- Von Neumann architecture: program and data in same memory
  - In addition, for x86: no distinction between executable and readable pages

**Attack #2: Pointer Variables**

1. Change a function pointer to point to the attack code
2. Any memory, on or off the stack, can be modified by a statement that stores a compromised value into the compromised pointer
Off-By-One Overflow

• Home-brewed range-checking string copy

```c
void notSoSafeCopy(char *input) {
    char buffer[512];
    int i;
    for (i=0; i<512; i++)
        buffer[i] = input[i];
}
void main(int argc, char *argv[]) {
    if (argc==2)
        notSoSafeCopy(argv[1]);
}
```

• 1-byte overflow: can’t change RET, but can change saved pointer to previous stack frame
  – On little-endian architecture, make it point into buffer
  – Caller’s RET will be read from the buffer!

---

Attack #3: Frame Pointer

Change the caller’s saved frame pointer to point to attacker-controlled memory. Caller’s return address will be read from this memory.
Defense #1: StackGuard

- Embed canaries (stack cookies) in stack frames and verify their integrity prior to function return
  - Any overflow of local variables will damage the canary

- Choose random canary string on program start
  - Attacker can’t guess what the value of canary will be
  - Example: /GS option in the .NET compiler

- Terminator canary: “\0”, newline, linefeed, EOF
  - String functions like strcpy won’t copy beyond “\0”

StackGuard Implementation

- StackGuard requires code recompilation

- Checking canary integrity prior to every function return causes a performance penalty
  - For example, 8% for Apache Web server

- StackGuard can be defeated
  - A single memory write where the attacker controls both the value and the destination is sufficient
Defeating StackGuard

• Suppose program contains `strcpy(dst,buf)` where attacker controls both dst and buf
  – Example: dst is a local pointer variable
• Can overwrite other function pointers
  – Exception handlers, virtual methods in C++

Defense #2: Address Space Layout Randomization (ASLR)

• Map shared libraries to a random location in process memory
  – Attacker does not know addresses of executable code

• Deployment
  – Windows Vista: 8 bits of randomness for DLLs
    • If aligned to 64K page in a 16MB region, then 256 choices
  – Linux (via PaX): 16 bits of randomness for libraries
  – More effective on 64-bit architectures

• Other randomization methods
  – Randomize system call ids or instruction set
Example: ASLR in Vista

Booting Vista twice loads libraries into different locations:

<table>
<thead>
<tr>
<th>Library</th>
<th>Base Address</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>ntlmssm.dll</td>
<td>0x6D7E0000</td>
<td>Microsoft Lan Manager</td>
</tr>
<tr>
<td>ntlmssm.dll</td>
<td>0x75370000</td>
<td>Windows NT MARTA provider</td>
</tr>
<tr>
<td>ntsh.dll</td>
<td>0x6F2C0000</td>
<td>Shell extensions for sharing</td>
</tr>
<tr>
<td>ole32.dll</td>
<td>0x76160000</td>
<td>Microsoft OLE for Windows</td>
</tr>
<tr>
<td>ntlmssm.dll</td>
<td>0x6DA90000</td>
<td>Microsoft Lan Manager</td>
</tr>
<tr>
<td>ntlmssm.dll</td>
<td>0x75660000</td>
<td>Windows NT MARTA provider</td>
</tr>
<tr>
<td>ntsh.dll</td>
<td>0x6C9D0000</td>
<td>Shell extensions for sharing</td>
</tr>
<tr>
<td>ole32.dll</td>
<td>0x763C0000</td>
<td>Microsoft OLE for Windows</td>
</tr>
</tbody>
</table>

Limitations:
- ASLR is only applied to images for which the dynamic-relocation flag is set
- Can still inject code on the stack and execute it

Defense #3: Code != Data

- Typical memory exploit involves code injection
  - Put malicious code at a predictable location in memory, usually masquerading as data
  - Trick vulnerable program into passing control to it
    - Overwrite saved EIP, function callback pointer, etc.

- Idea: prevent execution of untrusted code
  - Make stack and other data areas non-executable
    - Note: messes up useful functionality (e.g., Flash, JavaScript)
  - Digitally sign all code
  - Ensure that all control transfers are into a trusted, approved code image
**W⊕X / DEP**

- **Mark all writeable memory locations as non-executable**
  - Example: Microsoft’s Data Execution Prevention (DEP)
  - This blocks (almost) all code injection exploits
- **Hardware support**
  - AMD “NX” bit, Intel “XD” bit (in post-2004 CPUs)
  - Makes memory page non-executable
- **Widely deployed**
  - Windows (since XP SP2), Linux (via PaX patches), OS X (since 10.5)

---

**What Does W⊕X Not Prevent?**

- Can still corrupt stack ...
  - … or function pointers or critical data on the heap
- As long as the saved return address points into existing code, W⊕X protection will not block control transfer
- This is the basis of return-to-libc exploits
  - Overwrite return address with address of any library routine, arrange stack to look like arguments
- Does not look like a huge threat
  - Attacker cannot execute arbitrary code, especially if system() is not available
return-to-libc on Steroids

• Overwritten saved EIP need not point to the beginning of a library routine
• Any existing instruction in the code image is fine
  – Will execute the sequence starting from this instruction
• What if instruction sequence contains RET?
  – Execution will be transferred... to where?
  – Read the word pointed to by stack pointer (ESP)
    • Guess what? Its value is under attacker’s control! (why?)
  – Use it as the new value for EIP
    • Now control is transferred to an address of attacker’s choice!
  – Increment ESP to point to the next word on the stack

Chaining RETs for Fun and Profit
[Shacham et al.]

• Can chain together sequences ending in RET
  – Krahmer, “x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique” (2005)
• What is this good for?
• Answer [Shacham et al.]: everything
  – Turing-complete language
  – Build “gadgets” for load-store, arithmetic, logic, control flow, system calls
  – Attack can perform arbitrary computation using no injected code at all – return-oriented programming
Other Targets of Memory Exploits

- Configuration parameters
  - Example: directory names that confine remotely invoked programs to a portion of the file system
- Pointers to names of system programs
  - Example: replace the name of a harmless script with an interactive shell
    - This is not the same as return-to-libc
- Branch conditions in input validation code
- None of these exploits violate the integrity of the program’s control flow
  - Only original program code is executed!

Example: Web Server Security

- CGI scripts are executables on Web server that can be executed by remote user via a special URL
  - http://www.server.com/cgi-bin/SomeProgram
- Don’t want remote users executing arbitrary programs with the Web server’s privileges, need to restrict which programs can be executed
- CGI-BIN is the directory name which is always prepended to the name of the CGI script
  - If CGI-BIN is “/usr/local/httpd/cgi-bin”, the above URL will execute /usr/local/httpd/cgi-bin/SomeProgram
Exploiting Null HTTP Heap Overflow

- Null HTTPD had a heap overflow vulnerability
  - This enables the attacker to write an arbitrary value into a memory location of his choice

- Standard exploit: write address of attack code into the table containing addresses of library functions
  - Transfers control to attacker’s code next time the library function is called

- Alternative: overwrite the value of CGI-BIN

Null HTTP CGI-BIN Exploit

Server states

- Read CGI-BIN configuration
  - The configuration is just /etc/httpd/conf.d

- CGI-BIN configuration is now blank
  - without the string terminator ‘\0’

- CGI-BIN configuration is now blank
  - with the string terminator ‘\0’

- Server translates the file name as ‘/www’ and runs it using the string specified by the client as the standard input

- /opt/foot-private-file, writable only to the root, is removed

Client commands

- Send the first POST command to the server
  - to overwrite 2 bytes of CGI-BIN

- Send the second POST command to the server
  - to overwrite other 2 bytes of CGI-BIN

- Send the third POST command to run a shell command on the server:

  - POST /opt/summary http/1.1
  - Content-Length: 70
  - 'echo Content-type: text/plain
  - echo
  - echo
  - html/amp foot-private-file
  - echo
  - This will be the standard input string to /www on the server

- html/amp foot-private-file
Sources

• Various slides from Vitaly Shmatikov

Logistics

• Survey about the Network Security module
  – Log into Elms at http://elms.umd.edu
  – CMSC818V > Quizzes
    • Elms calls this a “quiz” but it’s not graded
    • Your responses are anonymized
  – Please answer all 8 questions
  – Comment only on the Network Security lectures
Review of Lecture

• What did we learn?
  – Memory corruption attacks: return address, function pointers, stack frames
  – Defenses: stack canaries, \( W \oplus X \), ASLR
  – Return oriented programming

• Paper discussion: “Automatic Patch-Based Exploit Generation”
  – MC2 Seminar by David Brumley: Nov 20, 5 pm, 1115 CSI
    • Register at https://talks.cs.umd.edu/talks/818
  – Discussion lead: John
  – Scribe: Octavian

• What’s next?
  – Worms and infection spreading