Memory Corruption

Basic Memory Corruption Attacks

Original slides were created by Prof. Dan Boneh
Memory corruption attacks

• **Attacker’s goal:**
  – Take over target machine (e.g. web server)
    • Execute arbitrary code on target by hijacking application control flow leveraging memory corruption

• Examples.
  – Buffer overflow attacks
  – Integer overflow attacks
  – Format string vulnerabilities
Example 1: buffer overflows

- Extremely common bug in C/C++ programs.
  - First major exploit: 1988 Internet Worm. fingerd.

\[ \approx 20\% \text{ of all vuln.} \]

Source: NVD/CVE
What is needed

• Understanding C functions, the stack, and the heap.
• Know how system calls are made
• The exec() system call

Attacker needs to know which CPU and OS used on the target machine:

– Our examples are for x86 running Linux or Windows
– Details vary slightly between CPUs and OSs:
  • Little endian vs. big endian (x86 vs. Motorola)
  • Stack Frame structure (Unix vs. Windows)
Linux process memory layout

- **user stack** at $0xC0000000$
- **shared libraries** at $0x40000000$
- **run time heap** at $0x08048000$
- **unused memory**

- **%esp** points to the stack
- **brk** points to the heap

**Loaded from exec**
Stack Frame

- arguments
- return address
- stack frame pointer
- exception handlers
- local variables

SP ➔

high ➔ Stack Growth ➒ low
What are buffer overflows?

Suppose a web server contains a function:

When `func()` is called stack looks like:

```c
void func(char *str) {
    char buf[128];
    strcpy(buf, str);
    do-something(buf);
}
```
What are buffer overflows?

What if *str is 136 bytes long?
After strcpy:

```c
void func(char *str) {
    char buf[128];
    strcpy(buf, str);
    do-something(buf);
}
```

Problem:
no length checking in strcpy()
Basic stack exploit

Suppose `*str` is such that after `strcpy` stack looks like:

Program P: `exec("/bin/sh")`

When `func()` exits, the user gets shell!

Note: attack code P runs in stack.
The NOP slide

Problem: how does attacker determine ret-address?

Solution: NOP slide

- Guess approximate stack state when `func()` is called
- Insert many NOPs before program P: `nop, xor eax,eax, inc ax`
Details and examples

• Some complications:
  – Program P should not contain the ‘\0’ character.
  – Overflow should not crash program before func() exists.

• (in)Famous remote stack smashing overflows:
    test.GetPrivateProfileString "file", [long string]
Many unsafe libc functions

`strcpy` (char *dest, const char *src)
`strcat` (char *dest, const char *src)
`gets` (char *s)
`scanf` (const char *format, ...) and many more.

- “Safe” libc versions `strncpy()`, `strncat()` are misleading
  - e.g. `strncpy()` may leave string unterminated.

- Windows C run time (CRT):
  - `strcpy_s` (*dest, DestSize, *src): ensures proper termination
Buffer overflow opportunities

• Exception handlers: (Windows SEH attacks)
  – Overwrite the address of an exception handler in stack frame.

• Function pointers: (e.g. PHP 4.0.2, MS MediaPlayer Bitmaps)
  – Overflowing buf will override function pointer.

• Longjmp buffers: longjmp(pos) (e.g. Perl 5.003)
  – Overflowing buf next to pos overrides value of pos.
Corrupting method pointers

- Compiler generated function pointers (e.g. C++ code)

After overflow of buf:

- NOP slide
- shell code
Finding buffer overflows

• To find overflow:
  – Run web server on local machine
  – Issue malformed requests (ending with “$$$$$$”)
    • Many automated tools exist (fuzzers, symbolic/concolic execution)
  – If web server crashes,
    search core dump for “$$$$$$” to find overflow location
• Construct exploit (not easy given latest defenses)
Memory Corruption

More Memory Corruption Attacks
More Corruption Opportunities

- **Integer overflows:** (e.g. MS DirectX MIDI Lib)

- **Double free:** double free space on heap
  - Can cause memory mgr to write data to specific location
  - Examples: CVS server

- **Use after free:** using memory after it is freed

- **Format string vulnerabilities**
Integer Overflows

Problem: what happens when int exceeds max value?

int m; (32 bits) short s; (16 bits) char c; (8 bits)

\[
\begin{align*}
c &= 0x80 + 0x80 = 128 + 128 \quad \Rightarrow \quad c &= 0 \\
s &= 0xff80 + 0x80 \quad \Rightarrow \quad s &= 0 \\
m &= 0xffffffff80 + 0x80 \quad \Rightarrow \quad m &= 0
\end{align*}
\]

Can this be exploited?
An example

```c
void func(char *buf1, *buf2, unsigned int len1, len2) {
    char temp[256];
    if (len1 + len2 > 256) {return -1} // length check
    memcpy(temp, buf1, len1); // cat buffers
    memcpy(temp+len1, buf2, len2);
    do-something(temp); // do stuff
}
```

What if \( \text{len1} = 0x80, \quad \text{len2} = 0xffffffff80 \) ?

\[ \Rightarrow \text{len1+len2} = 0 \]

Second memcpy() will overflow heap !!
Integer overflow exploit stats

Source: NVD/CVE
Format string bugs
int func(char *user) {
    fprintf(stderr, user);
}

Problem: what if *user = “%s%s%s%s%s%s%s” ??
  – Most likely program will crash: DoS.
  – If not, program will print memory contents. Privacy?
  – Full exploit using user = “%n”

Correct form: fprintf(stdout, “%s”, user);
Vulnerable functions

Any function using a format string.

Printing:
- printf, fprintf, sprintf, ...
- vprintf, vfprintf, vsprintf, ...

Logging:
- syslog, err, warn
Exploit

• Dumping arbitrary memory:
  – Walk up stack until desired pointer is found.
  – printf( “%08x.%08x.%08x.%08x|%s|”)  

• Writing to arbitrary memory:
  – printf( “hello %n”, &temp)  -- writes ‘6’ into temp.
  – printf( “%08x.%08x.%08x.%08x.%n”)
Memory Corruption

Platform Defenses
Preventing hijacking attacks

1. **Fix bugs:**
   - Audit software
     - Automated tools: Coverity, Prefast/Prefix.
   - Rewrite software in a type safe language (Java, ML)
     - Difficult for existing (legacy) code ...

2. Concede overflow, but **prevent code execution**

3. Add **runtime code** to detect overflows exploits
   - Halt process when overflow exploit detected
   - StackGuard, LibSafe, ...
Marking memory as non-execute \((W^X)\)

Prevent attack code execution by marking stack and heap as **non-executable**

- **NX-bit** on AMD Athlon 64, **XD-bit** on Intel P4 Prescott
  - NX bit in every Page Table Entry (PTE)

- **Deployment:**
  - Linux (via PaX project); OpenBSD
  - Windows: since XP SP2 (DEP)
    - Visual Studio: `/NXCompat[:NO]`

- **Limitations:**
  - Some apps need executable heap (e.g. JITs).
  - Does not defend against ‘**Return Oriented Programming**’ exploits
Examples: DEP controls in Windows

DEP terminating a program
Attack: Return Oriented Programming (ROP)

- Control hijacking without executing custom code

```
stack
  args
  ret-addr
  sfp
  local buf

libc.so
  exec()
  printf()
  "/bin/sh"
```
Response: randomization

- **ASLR**: (Address Space Layout Randomization)
  - Map shared libraries to random location in process memory
    ⇒ Attacker cannot jump directly to exec function
  - **Deployment**: (/DynamicBase)
    - *Windows 7*: 8 bits of randomness for DLLs
      - aligned to 64K page in a 16MB region ⇒ 256 choices
    - *Windows 8*: 24 bits of randomness on 64-bit processors

- **Other randomization methods**:
  - Sys-call randomization: randomize sys-call id’s
  - Instruction Set Randomization (ISR)
ASLR Example

Booting twice loads libraries into different locations:

<table>
<thead>
<tr>
<th>Library</th>
<th>Address 1</th>
<th>Address 2</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ntlanman.dll</td>
<td>0x6D7F0000</td>
<td>0x6DA90000</td>
<td>Microsoft® Lan Manager</td>
</tr>
<tr>
<td>ntmarta.dll</td>
<td>0x75370000</td>
<td>0x75660000</td>
<td>Windows NT MARTA provider</td>
</tr>
<tr>
<td>ntshrui.dll</td>
<td>0x6F2C0000</td>
<td>0x6D9D0000</td>
<td>Shell extensions for sharing</td>
</tr>
<tr>
<td>ole32.dll</td>
<td>0x76160000</td>
<td>0x763C0000</td>
<td>Microsoft OLE for Windows</td>
</tr>
</tbody>
</table>

Note: everything in process memory must be randomized
stack, heap, shared libs, base image

• Win 8 **Force ASLR**: ensures all loaded modules use ASLR
Defenses: Canary

• Many run-time checking techniques ...
  – we only discuss methods relevant to overflow protection

• Solution 1: StackGuard
  – Run time tests for stack integrity.
  – Embed “canaries” in stack frames and verify their integrity prior to function return.
Canary Types

• **Random canary:**
  – Random string chosen at program startup.
  – Insert canary string into every stack frame.
  – Verify canary before returning from function.
    • Exit program if canary changed. Turns potential exploit into DoS.
  – To corrupt, attacker must learn current random string.

• **Terminator canary:**  Canary = \{0, newline, linefeed, EOF\}
  – String functions will not copy beyond terminator.
  – Attacker cannot use string functions to corrupt stack.
Canaries are not full proof

- Canaries are an important defense tool, but do not prevent all control hijacking attacks:
  - Heap-based attacks still possible
  - Integer overflow attacks still possible
  - May not (depends on the implementation) prevent Exception Handling attacks
More methods ...

**StackShield**
- At function prologue, copy return address RET and SFP to “safe” location (beginning of data segment)
- Upon return, check that RET and SFP is equal to copy.
- Implemented as assembler file processor (GCC)

**Control Flow Integrity** (CFI)
- A combination of static and dynamic checking
  - Statically determine program control flow
  - Dynamically enforce control flow integrity
Memory Corruption

Advanced Attacks
Heap Spray Attacks

A reliable method for exploiting heap overflows
Heap-based control hijacking

- Compiler generated function pointers (e.g. C++ code)

Suppose vtable is on the heap next to a string object:
Heap-based control hijacking

• Compiler generated function pointers (e.g. C++ code)

After overflow of `buf` we have:

```
buf[256] vtable
```
A reliable exploit?

```javascript
<SCRIPT language="text/javascript">
    shellcode = unescape("%u4343%u4343%...");
    overflow-string = unescape("%u2332%u4276%...");
    cause-overflow(overflow-string);
    // overflow buf[
</SCRIPT>

Problem: attacker does not know where browser places shellcode on the heap
Heap Spraying [SkyLined 2004]

Idea:
1. use Javascript to spray heap with shellcode (and NOP slides)
2. then point vtable ptr anywhere in spray area
Javascript heap spraying

```
var nop = unescape("%u9090%u9090")
while (nop.length < 0x100000) nop += nop

var shellcode = unescape("%u4343%u4343%...";

var x = new Array()
for (i=0; i<1000; i++) {
    x[i] = nop + shellcode;
}
```

• Pointing func-ptr almost anywhere in heap will cause shellcode to execute.
Vulnerable buffer placement

- Placing vulnerable `buf[256]` next to object O:
  - By sequence of Javascript allocations and frees make heap look as follows:
  - Allocate vuln. buffer in Javascript and cause overflow
  - Successfully used against a Safari PCRE overflow [DHM’08]
Many heap spray exploits

<table>
<thead>
<tr>
<th>Date</th>
<th>Browser</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>11/2004</td>
<td>IE</td>
<td>IFRAME Tag BO</td>
</tr>
<tr>
<td>04/2005</td>
<td>IE</td>
<td>DHTML Objects Corruption</td>
</tr>
<tr>
<td>01/2005</td>
<td>IE</td>
<td>.ANI Remote Stack BO</td>
</tr>
<tr>
<td>07/2005</td>
<td>IE</td>
<td>javaprxy.dll COM Object</td>
</tr>
<tr>
<td>03/2006</td>
<td>IE</td>
<td>createTextRange RE</td>
</tr>
<tr>
<td>09/2006</td>
<td>IE</td>
<td>VML Remote BO</td>
</tr>
<tr>
<td>03/2007</td>
<td>IE</td>
<td>ADODB Double Free</td>
</tr>
<tr>
<td>09/2006</td>
<td>IE</td>
<td>WebViewFolderIcon setSlice</td>
</tr>
<tr>
<td>09/2005</td>
<td>FF</td>
<td>0xAD Remote Heap BO</td>
</tr>
<tr>
<td>12/2005</td>
<td>FF</td>
<td>compareTo() RE</td>
</tr>
<tr>
<td>07/2006</td>
<td>FF</td>
<td>Navigator Object RE</td>
</tr>
<tr>
<td>07/2008</td>
<td>Safari</td>
<td>Quicktime Content-Type BO</td>
</tr>
</tbody>
</table>

[RLZ’08]

• Improvements: Heap Feng Shui [S’07]
  – Reliable heap exploits on IE without spraying
  – Gives attacker full control of IE heap from Javascript
Defenses

- Protect heap function pointers
- Better browser architecture:
  - Store JavaScript strings in a separate heap from browser heap
- OpenBSD heap overflow protection:
  - Nozzle \textsuperscript{[RLZ'08]}: detect sprays by prevalence of code on heap

\begin{itemize}
  \item Prevents cross-page overflows
\end{itemize}
References on heap spraying


[2] Engineering Heap Overflow Exploits with JavaScript

   by P. Ratanaworabhan, B. Livshits, and B. Zorn

[4] Interpreter Exploitation: Pointer inference and JiT spraying,
   by Dion Blazakis