Control Hijacking Attacks

Note: project 1 is out

Section this Friday 2pm (Skilling 090)
Control hijacking attacks

Attacker’s goal:
- Take over target machine (e.g. web server)
  - Execute arbitrary code on target by hijacking application control flow

This lecture: three examples.
- Buffer overflow attacks
- Integer overflow attacks
- Format string vulnerabilities

Project 1: Build exploits
1. Buffer overflows

Extremely common bug.
- First major exploit: 1988 Internet Worm. fingerd.

Developing buffer overflow attacks:
- Locate buffer overflow within an application.
- Design an exploit.

Source: NVD/CVE

≈20% of all vuln.
2005-2007: ≈ 10%
What is needed

- Understanding C functions and the stack
- Some familiarity with machine code
- Know how systems calls are made
- The exec() system call

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

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

- user stack
- shared libraries
- run time heap
- unused

%esp
brk

Loaded from exec

Addresses:
- 0x8048000
- 0xC0000000
- 0x40000000
- 0x08048000
- 0

Stack Frame

Parameters
Return address
Stack Frame Pointer
Local variables

SP
Stack Growth
What are buffer overflows?

Suppose a web server contains a function:

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

When the function is invoked the stack looks like:

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

What if `*str` is 136 bytes long? After `strcpy`:
Basic stack exploit

Problem: no range checking in \texttt{strcpy}().

Suppose \*\texttt{str} is such that after \texttt{strcpy} stack looks like:

\begin{center}
\begin{tabular}{c|c|c|c}
*str & ret & Code for P \\
\hline
\end{tabular}
\end{center}

Program P: \texttt{exec( "/bin/sh" )}
(exact shell code by Aleph One)

When \texttt{func()} exits, the user will be given a shell!

Note: attack code runs \textit{in stack}.

To determine \texttt{ret} guess position of stack when \texttt{func()} is called
Many unsafe C lib functions

strcpy (char *dest, const char *src)
strcat (char *dest, const char *src)
gets (char *s)
scanf (const char *format, ...)

“Safe” versions strcpy(), strcat() are misleading
- strncpy() may leave buffer unterminated.
- strncpy(), strcat() encourage off by 1 bugs.
Exploiting buffer overflows

Suppose web server calls `func()` with given URL.
- Attacker sends a 200 byte URL. Gets shell on web server

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

Sample remote buffer overflows of this type:
- (2005) Overflow in MIME type field in MS Outlook.
- (2005) Overflow in Symantec Virus Detection

```vbnet
Set test = CreateObject("Symantec.SymVAFileQuery.1")
test.GetPrivateProfileString "file", [long string]
```
Control hijacking opportunities

- Stack smashing attack:
  - Override return address in stack activation record by overflowing a local buffer variable.

- 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.
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 \texttt{buf} we have:

\begin{itemize}
\item \texttt{buf[256]}
\item \texttt{vtable}
\end{itemize}
Other types of overflow attacks

- **Integer overflows:** (e.g. MS DirectX MIDI Lib) Phrack60

  ```c
  void func(int a, char v) {
    char buf[128];
    init(buf);
    buf[a] = v;
  }
  ```

  - Problem: `a` can point to `ret-addr` on stack.

- **Double free:** double free space on heap.
  - Can cause mem mgr to write data to specific location
  - Examples: CVS server
Integer overflow stats

Source: NVD/CVE
Finding buffer overflows

- To find overflow:
  - Run web server on local machine
  - Issue requests with long tags
    All long tags end with “$$$$$$”
  - If web server crashes,
    search core dump for “$$$$$$” to find overflow location

- Many automated tools exist (called fuzzers – next lecture)

- Then use disassemblers and debuggers (e.g. IDA-Pro) to construct exploit
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 overflow code execution by marking stack and heap segments 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)
    - Boot.ini: `/noexecute=OptIn` or `AlwaysOn`

- Limitations:
  - Some apps need executable heap (e.g. JITs).
  - Does not defend against `return-to-libc` exploit
Examples: DEP controls in Vista

DEP terminating a program
Attack: return to libc

Control hijacking without executing code

Generalization: can generate arbitrary programs using return oriented programming
Response: randomization

**ASLR:** (Address Space Layout Randomization)
- Map shared libraries to random location in process memory
  ⇒ Attacker cannot jump directly to exec function
- Deployment:
  - **Windows Vista:** 8 bits of randomness for DLLs
    - aligned to 64K page in a 16MB region ⇒ 256 choices
  - **Linux** (via PaX): 16 bits of randomness for libraries
  - More effective on 64-bit architectures

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

Booting Vista twice loads libraries into different locations:

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

Note: ASLR is only applied to images for which the dynamic-relocation flag is set.
Attack: JIT spraying

Idea:
1. Force Javascript JIT to fill heap with executable shellcode
2. then point SFP anywhere in spray area
Run time checking
Run time checking: StackGuard

Many 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:**
- Choose random string at program startup.
- Insert canary string into every stack frame.
- Verify canary before returning from function.
- To corrupt random canary, 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.
StackGuard (Cont.)

- StackGuard implemented as a GCC patch.
  - Program must be recompiled.

- Minimal performance effects: 8% for Apache.

- Note: Canaries don’t offer fullproof protection.
  - Some stack smashing attacks leave canaries unchanged

- Heap protection: PointGuard.
  - Protects function pointers and setjmp buffers by encrypting them: XOR with random cookie
  - More noticeable performance effects
StackGuard variants - ProPolice

- **ProPolice (IBM)** - gcc 3.4.1. (-fstack-protector)
  - Rearrange stack layout to prevent ptr overflow.

```
Stack Growth

String Growth

<table>
<thead>
<tr>
<th></th>
<th>No arrays or pointers</th>
</tr>
</thead>
<tbody>
<tr>
<td>ret addr</td>
<td></td>
</tr>
<tr>
<td>SFP</td>
<td></td>
</tr>
<tr>
<td><strong>CANARY</strong></td>
<td></td>
</tr>
<tr>
<td>arrays</td>
<td></td>
</tr>
<tr>
<td>local variables</td>
<td></td>
</tr>
</tbody>
</table>
```

- Ptrs, but no arrays
Compiler /GS option:
- Combination of ProPolice and Random canary.
- Triggers UnHandledException in case of Canary mismatch to shutdown process.

Litchfield vulnerability report
- Overflow overwrites exception handler
- Redirects exception to attack code

```assembly
mov    eax, dword ptr [__security_cookie]
xor    eax, ebp
mov    dword ptr [ebp-8], eax
...
mov    ecx, dword ptr [ebp-8]
xor    ecx, ebp
call   __security_check_cookie@4
```
Run time checking: Libsafe

Solution 2: Libsafe (Avaya Labs)

- Dynamically loaded library (no need to recompile app.)
- Intercepts calls to `strcpy` (dest, src)
  - Validates sufficient space in current stack frame:
    \[ |\text{frame-pointer} - \text{dest}| > \text{strlen(src)} \]
  - If so, does `strcpy`, otherwise, terminates application
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
Format string bugs
Format string problem

```c
int func(char *user) {
    fprintf(stdout, user);
}
```

Problem: what if `user = "%s%s%s%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:

```c
int func(char *user) {
    fprintf(stdout, "%s", user);
}
```
History

First exploit discovered in June 2000.

Examples:
- wu-ftp 2.* : remote root
- Linux rpc.statd: remote root
- IRIX telnetd: remote root
- BSD chpass: local root
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" )`
Overflow using format string

```c
char errmsg[512], outbuf[512];

sprintf (errmsg, "Illegal command: %400s", user);
...  
sprintf( outbuf, errmsg );
```

- What if user = "%500d <nops> <shellcode>"
  - Bypass "%400s" limitation.
  - Will overflow outbuf.
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:

```
buf[256]  vtable  object T
```
Heap-based control hijacking

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

- After overflow of \texttt{buf} we have:

\begin{itemize}
  \item \texttt{buf}[256]
  \item \texttt{vtable}
  \item \texttt{ptr} \quad \texttt{data}
  \item \texttt{shell code}
  \item \texttt{object T}
\end{itemize}
A reliable exploit?

```javascript
<SCRIPT language="text/javascript">
let shellcode = unescape('%u4343%u4343%...');
let 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

Idea:

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

```javascript
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 \texttt{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>createTextRang 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>

**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 (e.g. PointGuard)

- Better browser architecture:
  - Store JavaScript strings in a separate heap from browser heap

- OpenBSD heap overflow protection:
  - Prevents cross-page overflows

- Nozzle [RLZ'08]: detect sprays by prevalence of code on heap
References on heap spraying


[2] Engineering Heap Overflow Exploits with JavaScript
M. Daniel, J. Honoroff, and C. Miller, WooT 2008

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

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