Lecture 8: Control hijacking attacks
Control hijacking attacks

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

**Examples in this lecture:**
- Buffer overflow attacks
- Integer overflow attacks
- Format string vulnerabilities

**Various countermeasures**
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

- **%esp**
- **brk**
- **Loaded from exec**
- **user stack**
- **shared libraries**
- **run time heap**
- **unused**

Memory addresses:
- 0x08048000
- 0x40000000
- 0xC0000000
- 0x08048000
- 0
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:

- str
- ret-addrs
- sfp
- buf
- top of stack

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

- *str
- ret
- str
- top of stack
Basic stack exploit

- Problem: no range checking in `strcpy()`.
- Suppose `*str` is such that after `strcpy` stack looks like:

```
*str        ret        NOP slide   code for P
```

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

- When `func()` exits, the user will be given a shell!
- Note: attack code runs in stack.

To determine `ret` guess position of stack when `func()` is called
Terminology

- **Bug**
  might be a
- **Vulnerability**
  for which someone will write an
- **Exploit**
  that hijacks control and runs
- **Shellcode**
  that typically installs a
- **Rootkit**
  that “Own” the computer and hides the traces.
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 `strncpy()`, `strncat()` are misleading

- `strncpy()` may leave buffer unterminated.
- `strncpy()`, `strncat()` 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

    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 buf:

- NOP slide
- shell code

Object T

- ptr
- data

vtable

FP1
FP2
FP3

method #1
method #2
method #3
Other types of overflow attacks

- **Integer overflows:** (e.g. MS DirectX MIDI Lib) Phrack60
  ```
  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
      - Visual Studio: /NXCompat[:NO]

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

Data Execution Prevention (DEP) helps protect against damage from viruses and other security threats. How does it work?

Turn on DEP for essential Windows programs and services only.

Turn on DEP for all programs and services except those I select.

Your computer's processor supports hardware-based DEP.

To help protect your computer, Windows has closed this program.

Name: Windows Explorer
Publisher: Microsoft Corporation

Data Execution Prevention helps protect against damage from viruses and other security threats. What should I do?

DEP terminating a program
Attack: return to libc

Control hijacking without executing code

For shellcode like `exe("/bin/sh")`, don’t worry about the stack frame pointer (`sfp`). The x86 function exit sequence is:

```
mov %ebp, %esp
pop %ebp
ret
```

When the attacked functions returns, it will load a corrupted `sfp` value into `%ebp`, but `%esp` will be correctly restored to point to `ret-addr`, so the jump to libc will work. The libc functions read their arguments relative to this (uncorrupted) `%esp`. The corrupted `sfp` matters only if/when libc returns.

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: (/DynamicBase)
    - 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)
  - Reversed stack direction
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>ntmarta.dll</td>
<td>0x75370000</td>
<td>Windows NT MARTA provider</td>
</tr>
<tr>
<td>ntshrui.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>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Library</th>
<th>Offset</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ntlanman.dll</td>
<td>0x6DA90000</td>
<td>Microsoft® Lan Manager</td>
</tr>
<tr>
<td>ntmarta.dll</td>
<td>0x75660000</td>
<td>Windows NT MARTA provider</td>
</tr>
<tr>
<td>ntshrui.dll</td>
<td>0x6D9D0000</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>

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
  - Less effective, more noticeable performance effects
StackGuard variants - ProPolice

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

String Growth:
- args
- ret addr
- SFP
- CANARY
  - No arrays or pointers

Stack Growth:
- arrays
- local variables
  - Ptrs, but no arrays
MS Visual Studio /GS [2003]

Compiler /GS option:
- Combination of ProPolice and Random canary.
- Triggers UnHandledException in case of Canary mismatch to shutdown process.

```
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
```

- Litchfield vulnerability report
  - Overflow overwrites exception handler
  - Redirects exception to attack code
  - /SafeSEH: only call pre-designated exception handler
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(stderr, user);
}
```

**Problem:** what if \texttt{user = \textasciitilde{s}\textasciitilde{s}\textasciitilde{s}\textasciitilde{s}\textasciitilde{s}\textasciitilde{s}\textasciitilde{s}\textasciitilde{s}\textasciitilde{s}\textasciitilde{s}}\

- Most likely program will crash: DoS.
- If not, program will print memory contents. Privacy?
- Full exploit using \texttt{user = \textasciitilde{\%n}}

Correct form:

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

- First exploit discovered in June 2000.

Examples:
- `wu-ftp"d 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

らしい メモリを任意に記憶:
- Walk up stack until desired pointer is found.
- printf( "%08x.%08x.%08x.%08x|%s|"")

書き込み メモリを任意に記憶:
- 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.