System Security

• Offense/Attack
  – Break the system to gain resources with respect to confidentiality (something you are not supposed to access) and integrity (something you are not supposed to modify)

• Defense/Protection
  – Protect information and system resources with respect to confidentiality, integrity, and availability (defending such as Denial of service attack)

Attack Techniques

• Find memory vulnerabilities
  – Buffer overflow, integer overflow, and etc.

• Develop exploits
  – Memory exploits, shellcode, heapspray, and etc.

• Create malware
  – Obfuscation
  – Packing (encryption)
Defense Techniques

• Stop exploits
  – Architecture, hardware
  – OS, loader, linker
  – Compiler

• Analyze malware (Reverse Engineering)
  – Unpacking
  – De-transformation

• Ensure Availability

Understanding OS Kernels

• Process Management
  – Creation/Running/Exit
  – Process address sapce
  – Context switch

• Virtual Memory
  – paging

• File System and Disk Data Management
  – EXT2/EXT3, NTFS
  – Proc
Understanding Program Runtime

- Unveiling Program Execution
  - Stack, Heap, Global
  - Control flow
- System Loader
  - ld-linux.so, ld_preload
  - DLL Injection
- Dynamic Linker
  - PLT/GOT
  - Online-patching

Understanding Program Analysis

- Data Flow Analysis
  - Data dependency
  - Taint analysis
  - Point-to analysis
- Control Flow Analysis
  - Control flow graph
  - Call graph
  - Control dependency
- Path-(in)sensitive
- Context-(in)sensitive
Vulnerability

• Control flow hijacking
  – Buffer overflow
  – Integer overflow
  – Format string

• Data manipulation

Control Hijacking

Basic 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.
  – 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.

≈20% 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

![Diagram of Linux process memory layout]

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

- 0x08048000
- 0x40000000
- 0xC0000000
- 0xC0000000
- 0x08048000
- 0

00000000
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 `func()` is called stack looks like:

```
argument: str
return address
stack frame pointer
char buf[128]
```
What are buffer overflows?

What if \*str is 136 bytes long?

After \texttt{strcpy}:

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

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

Basic stack exploit

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

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

When \texttt{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)

```
buf[128] FuncPtr
```

  - 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:

Finding buffer overflows

- To find overflow:
  - Run web server on local machine
  - Issue malformed requests (ending with "$\$\$\$\$" )
    - Many automated tools exist (called fuzzers – next module)
  - If web server crashes,
    search core dump for "$\$\$\$\$" to find overflow location

- Construct exploit (not easy given latest defenses)
Control Hijacking

More Control Hijacking Attacks

More Hijacking 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

- **Format string vulnerabilities**
# Integer Overflows

Problem: what happens when int exceeds max value?

<table>
<thead>
<tr>
<th>Type</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>int m;</td>
<td>(32 bits)</td>
</tr>
<tr>
<td>short s;</td>
<td>(16 bits)</td>
</tr>
<tr>
<td>char c;</td>
<td>(8 bits)</td>
</tr>
</tbody>
</table>

\[
c = 0x80 + 0x80 = 128 + 128 \Rightarrow c = 0
\]

\[
s = 0xff80 + 0x80 \Rightarrow s = 0
\]

\[
m = 0xffffffff80 + 0x80 \Rightarrow m = 0
\]

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

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

Second memcpy() will overflow heap!!
Format string bugs

Format string problem

```c
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:**
```c
    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")
Control Hijacking

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-to-libc` exploits

---

Examples: DEP controls in Windows

![Examples](image.png)

DEP terminating a program
Attack: return to libc

- Control hijacking without executing code

<table>
<thead>
<tr>
<th>Stack</th>
<th>libc.so</th>
</tr>
</thead>
<tbody>
<tr>
<td>args</td>
<td>exec()</td>
</tr>
<tr>
<td>ret-addr</td>
<td>printf()</td>
</tr>
<tr>
<td>sfp</td>
<td>“/bin/sh”</td>
</tr>
<tr>
<td>local buf</td>
<td></td>
</tr>
</tbody>
</table>

Response: randomization

- **ASLR**: (Address Space Layout Randomization)
  - Map shared libraries to rand 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
    - **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>DLL</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>ntdsnt.dll</td>
<td>0x75370000</td>
<td>Windows NT MARTA provider</td>
</tr>
<tr>
<td>ntdsnt.dll</td>
<td>0x8F2C0000</td>
<td>Shell extensions for sharing</td>
</tr>
<tr>
<td>ole32.dll</td>
<td>0x87615000</td>
<td>Microsoft OLE for Windows</td>
</tr>
<tr>
<td>ntlanman.dll</td>
<td>0x6DA90000</td>
<td>Microsoft® Lan Manager</td>
</tr>
<tr>
<td>ntdsnt.dll</td>
<td>0x75660000</td>
<td>Windows NT MARTA provider</td>
</tr>
<tr>
<td>ntdsnt.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: everything in process memory must be randomized
stack, heap, shared libs, image

• Win 8 Force ASLR: ensures all loaded modules use ASLR

More attacks: JIT spraying

Idea:
1. Force Javascript JIT to fill heap with executable shellcode
2. then point SFP anywhere in spray area
Control Hijacking

Run-time Defenses

Run time checking: StackGuard

• 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.

StackGuard (Cont.)

• StackGuard implemented as a GCC patch
  – Program must be recompiled

• Minimal performance effects: 8% for Apache

• Note: Canaries do not provide full protection
  – Some stack smashing attacks leave canaries unchanged

• Heap protection: PointGuard
  – Protects function pointers and setjmp buffers by encrypting them:
    e.g. XOR with random cookie
  – Less effective, more noticeable performance effects
StackGuard enhancements: ProPolice

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

---

**String Growth**

- args
- ret addr
- SFP
- CANARY

**Stack Growth**

- local string buffers
- local non-buffer variables
- copy of pointer args

---

MS Visual Studio /GS [since 2003]

Compiler /GS option:
- Combination of ProPolice and Random canary.
- If cookie mismatch, default behavior is to call _exit(3)

---

**Function prolog:**

```
sub esp, 8    // allocate 8 bytes for cookie
mov eax, DWORD PTR ___security_cookie
xor eax, esp   // xor cookie with current esp
mov DWORD PTR [esp+8], eax   // save in stack
```

**Function epilog:**

```
mov ecx, DWORD PTR [esp+8]
xor ecx, esp
call @__security_check_cookie@4
add esp, 8
```

---

Enhanced /GS in Visual Studio 2010:
- /GS protection added to all functions, unless can be proven unnecessary
**Evading /GS with exception handlers**

- When exception is thrown, dispatcher walks up exception list until handler is found (else use default handler)

After overflow: handler points to attacker’s code

exception triggered ⇒ control hijack

Main point: exception is triggered before canary is checked
Defenses: SAFESEH and SEHOP

- **SAFESEH**: linker flag
  - Linker produces a binary with a table of safe exception handlers
  - System will not jump to exception handler not on list

- **SEHOP**: platform defense (since win vista SP1)
  - Observation: SEH attacks typically corrupt the “next” entry in SEH list.
  - SEHOP: add a dummy record at top of SEH list
  - When exception occurs, dispatcher walks up list and verifies dummy record is there. If not, terminates process.

Summary: 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
  - *GS by itself does not prevent Exception Handling attacks* (also need SAFESEH and SEHOP)
What if can’t recompile: 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

How robust is Libsafe?

strcpy() can overwrite a pointer between buf and sfp.
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

---

Control Hijacking

**Advanced Hijacking 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)

![Diagram of heap-based control hijacking]

- After overflow of `buf` we have:

![Diagram after heap overflow]

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

![Diagram of heap-based control hijacking with shellcode]

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
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:

![Diagram of heap placement](image)

- 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>javaapxy.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>OXML 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
(partial) 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:
  - Nozzle [RLZ'08]: detect sprays by prevalence of code on heap

prevents cross-page overflows