Acknowledgments: Lecture slides are from the Computer Security course taught by Dan Boneh at Stanford University. When slides are obtained from other sources, a reference will be noted on the bottom of that slide. A full list of references is provided on the last slide.
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.

Source: web.nvd.nist.gov
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)
Memory Organization
The Stack grows down towards lower addresses.

- Stack grows down
- Heap grows up

- Memory
- Program text
- Shared libs
- Data
  ...

User stack: 0x00000000

Shared libraries: 0xC0000000 (3GB)

Run time heap: 0x00000000

%esp and brk references.
Variables

• On the stack
  – Local variables
  – Lifetime: stack frame

• On the heap
  – Dynamically allocated via new/malloc/etc.
  – Lifetime: until freed

[Brumley]
Procedures

• Procedures are not native to assembly
• Compilers *implement* procedures
  – On the stack
  – Following the call/return stack discipline
Procedures/Functions

• We need to address several issues:
  1. How to allocate space for local variables
  2. How to pass parameters
  3. How to pass return values
  4. How to share 8 registers with an infinite number of local variables

• A stack frame provides space for these values
  – Each procedure invocation has its own stack frame
  – Stack discipline is LIFO
    • If procedure A calls B, B’s frame must exit before A’s
Function Call Chain

orange(…)
{
  ...
  red()
  ...
}

red(…)
{
  ...
  green()
  ...
  green()
}

green(…)
{
  ...
  green()
  ...
}
Frame for
- locals
- pushing parameters
- temporary space

Call to red
"pushes" new frame

When green returns it
"pops" its frame

Function Call Chain

orange
↓
red
↓
green
↓
green
↓
...

[Brumley]
int orange(int a, int b) {
    char buf[16];
    int c, d;
    if(a > b)
        c = a;
    else
        c = b;
    d = red(c, buf);
    return d;
}
cdecl - the default for Linux & gcc

int orange(int a, int b)
{
  char buf[16];
  int c, d;
  if(a > b)
    c = a;
  else
    c = b;
  d = red(c, buf);
  return d;
}
When *orange* attains control,

1. return address has already been pushed onto stack by caller
When orange attains control,
1. return address has already been pushed onto stack by caller
2. own the frame pointer
   - push caller’s ebp
   - copy current esp into ebp
   - first argument is at ebp+8
When orange attains control,
1. return address has already been pushed onto stack by caller
2. own the frame pointer
   - push caller’s ebp
   - copy current esp into ebp
   - first argument is at ebp+8
3. save values of other callee-save registers if used
   - edi, esi, ebx: via push or mov
   - esp: can restore by arithmetic
When orange attains control,

1. return address has already been pushed onto stack by caller
2. own the frame pointer
   - push caller’s ebp
   - copy current esp into ebp
3. save values of other callee-save registers if used
   - edi, esi, ebx: via push or mov
   - esp: can restore by arithmetic
4. allocate space for locals
   - subtracting from esp
   - “live” variables in registers, which on contention, can be “spilled” to stack space
For *caller orange* to call *callee red*,

<table>
<thead>
<tr>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
</tr>
<tr>
<td>a</td>
</tr>
<tr>
<td>return addr</td>
</tr>
<tr>
<td>caller’s ebp</td>
</tr>
<tr>
<td>callee-save</td>
</tr>
<tr>
<td>locals</td>
</tr>
<tr>
<td>(buf, c, d ≥ 24 bytes if stored on stack)</td>
</tr>
</tbody>
</table>
For *caller* orange to call *callee* red,
1. push any caller-save registers if their values are needed after red returns
   - eax, edx, ecx

[Diagram showing stack layout with `%ebp` and `%esp` pointers]

[Brumley]
For *caller orange* to call *callee red*,

1. push any caller-save registers if their values are needed after *red* returns
   - eax, edx, ecx

2. push arguments to *red* from right to left (reversed)
   - from callee’s perspective, argument 1 is nearest in stack
For **caller orange** to call **callee red**,

1. push any caller-save registers if their values are needed after **red** returns
   - eax, edx, ecx
2. push arguments to **red** from right to left (reversed)
   - from callee’s perspective, argument 1 is nearest in stack
3. push return address, i.e., the *next* instruction to execute in **orange** after **red** returns
For caller orange to call callee red,
1. push any caller-save registers if their values are needed after red returns
   – eax, edx, ecx
2. push arguments to red from right to left (reversed)
   – from callee’s perspective, argument 1 is nearest in stack
3. push return address, i.e., the next instruction to execute in orange after red returns
4. transfer control to red
   – usually happens together with step 3 using call

[Brumley]
When red attains control,
1. return address has already been pushed onto stack by orange
When red attains control,

1. return address has already been pushed onto stack by orange
2. own the frame pointer

[Diagram of stack layout with variables and memory locations labeled]

[Brumley]
When red attains control,
1. return address has already been pushed onto stack by orange
2. own the frame pointer
3. … (red is doing its stuff) …
When red attains control,
1. return address has already been pushed onto stack by orange
2. own the frame pointer
3. … (red is doing its stuff) …
4. store return value, if any, in eax
5. deallocate locals
   - adding to esp
6. restore any callee-save registers
When red attains control,
1. return address has already been pushed onto stack by orange
2. own the frame pointer
3. … (red is doing its stuff) …
4. store return value, if any, in eax
5. deallocate locals
   - adding to esp
6. restore any callee-save registers
7. restore orange’s frame pointer
   - pop %ebp
When red attains control,
1. return address has already been pushed onto stack by orange
2. own the frame pointer
3. … (red is doing its stuff) …
4. store return value, if any, in eax
5. deallocate locals
   - adding to esp
6. restore any callee-save registers
7. restore orange’s frame pointer
   - pop %ebp
8. return control to orange
   - ret
   - pops return address from stack and jumps there
When orange regains control,
When *orange* regains control,
1. clean up arguments to *red*
   - adding to esp
2. restore any caller-save registers
   - pops
3. …
Linux process memory layout

- **User Stack**: $0xC0000000$
- **Shared Libraries**: $0x40000000$
- **Run Time Heap**: $0x08048000$
- **Unused**: $0x0$

- `%esp` points to the user stack
- `brk` points to the run time heap
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:  \texttt{exec("/bin/sh")}

(exact shell code by Aleph One)

When func() exits, the user gets shell!

Note: attack code P runs \textit{in stack}. 
The NOP slide

Problem: how does attacker determine return 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 $\text{func()}$ exists.

• (in)Famous **remote** stack smashing overflows:
  – Overflow in Windows animated cursors (ANI).
    \text{LoadAniIcon()}
  – Past overflow in Symantec virus detection
    \text{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):
  – `strncpy_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:

Object T

ptr
data

code

vtable

FP1
FP2
FP3

• method #1
• method #2
• method #3

buf[256]

ptr
data

vtable

object T

NOP

slide

shell

Dan Boneh
Finding buffer overflows

• To find overflow:
  – Run web server on local machine
  – Issue malformed requests (ending with “$$$$$$”)
    • Many automated tools exist (called fuzzers - next week)
  – If web server crashes,
    search core dump for “$$$$$$” 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

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

• **Format string vulnerabilities**
Integer Overflows
(see Phrack 60)

Problem: what happens when int exceeds max value?

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

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

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

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

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

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

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

Source: NVD/CVE
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: `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 (DEP)

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 code
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
  - Windows 8: 24 bits of randomness on 64-bit processors

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

Booting twice loads libraries into different locations:

<table>
<thead>
<tr>
<th>DLL</th>
<th>Base Address</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>DLL</th>
<th>Base Address</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: everything in process memory must be randomized
stack, heap, shared libs, base image

• Win 8 Force ASLR: ensures all loaded modules use ASLR
Control Hijacking Defenses

Hardening the executable
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
  - More noticeable performance effects
StackGuard enhancements: ProPolice

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

![Stack layout diagram]

- Protects pointer args and local pointers from a buffer overflow
- No protection for local string buffers or local non-buffer variables
- Protects copy of pointer args
- Stack growth from bottom to top
- String growth from left to right
MS Visual Studio /GS [since 2003]

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

Function prolog:
- \texttt{sub esp, 4} // allocate 4 bytes for cookie
- \texttt{mov eax, DWORD PTR ___security_cookie}
- \texttt{xor eax, esp} // xor cookie with current esp
- \texttt{mov DWORD PTR [esp], eax} // save in stack

Function epilog:
- \texttt{mov ecx, DWORD PTR [esp]}
- \texttt{xor ecx, esp}
- \texttt{call @__security_check_cookie@4}
- \texttt{add esp, 4}

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

Dan Boneh
/GS stack frame

- **String Growth**
  - args
  - ret addr
  - SFP
  - exception handlers
  - CANARY

- **Stack Growth**
  - local string buffers
  - local non-buffer variables
  - copy of pointer args

- Canary protects ret-addr and exception handler frame

- Pointers, but no arrays
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: SAFESEEH and SEHOP

- **SAFESEEH**: 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
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 Flow Guard (CFG) (Windows 10)

Poor man’s version of CFI:

- Protects indirect calls by checking against a bitmask of all valid function entry points in executable

```assembly
rep stosd
mov  esi, [esi]  ; Target
mov  ecx, esi
push 1
call @guard_check_icall@4 ; _guard_check_icall(x)
call esi
add  esp, 4
xor  eax, eax
```

ensures target is the entry point of a function
Control Flow Guard (CFG) (Windows 10)

Poor man’s version of CFI:

- Protects indirect calls by checking against a bitmask of all valid function entry points in executable.
- Ensures target is the entry point of a function.
- Does not prevent attacker from causing a jump to a valid wrong function.
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:

```
buf[256]  vtable  ptr  data
```

Object T
Heap-based control hijacking

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

• After overflow of `buf` we have:

```
ptr
data

Object T

FP1
FP2
FP3
vtable

method #1
method #2
method #3

shell
code

buf[256]
vtable

ptr
data

object T
```
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

```plaintext
buf[256] vtable 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 heap spraying

var nop = unescape(“%u9090%u9090”)
while (nop.length < 0x100000) nop += nop

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

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.
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>DIIIHTML 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>createTextSpan 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>ADO DB Double Free</td>
</tr>
<tr>
<td>09/2006</td>
<td>IE</td>
<td>WebViewFolderIco 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>07/2005</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

[RLZ’08]
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
References on heap spraying


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

• Acknowledgments: Some of the slides are fully or partially obtained from other sources. Reference is noted on the bottom of each slide, when the content is fully obtained from another source. Otherwise a full list of references is provided on the last slide.
