Logistics

- Assignment 1 coming out after this lecture
- Sign up for groups
- Waitlists cleared
  - Only registrar constraints prevent enrollement

- Everyone is enrolled in Piazza and Autolab
  - Autolab: milkshark.ics.cs.cmu.edu
  - Regardless of waitlist status
Control Hijacking Attacks

18-732
Spring 2015
Lujo Bauer
Today

- Much of software security is motivated by common attacks
- Today: control hijacking attacks + some defenses
- Thursday: hardware and software principles to mitigate effects of attacks
- Later: even more defenses
  - Static & dynamic analyses, language-based techniques
Today

- Much of software security is motivated by common attacks
- Today: control hijacking attacks + some defenses

Goals:
- Understand in general terms what we’re up against
- Understand key characteristics of attacks
- Understand vulnerabilities in software that make attacks possible

... so that we can better defend against attacks
Control hijacking attacks

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

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

- Extremely common bug
  - First major exploit: 1988 Internet Worm (fingerd)
  - Nat’l Vuln DB: 9 buffer overflow vulns. reported in the past week

% of reported vulnerabilities that match “buffer overflow”

[National Vulnerability Database
void main()
{
    func(1, 2, 3);
}

void func(int a, int b, int c)
{
    char buffer1[5];
    char buffer2[10];
}

Function call: push
- Parameters, current %EIP as return address, saved frame pointer, local vars
void main()
{
    func(1,2,3);
}

void func(int a, int b, int c)
{
    char buffer1[5];
    char buffer2[10];
}

Function return: *pop*

- Activation record,
  update frame pointer
  using ebp, update %EIP using RET
Stack frame

- Parameters
- Return address
- Saved frame pointer
- Local variables

Stack growth

- Larger memory addresses
- Smaller memory addresses

SP
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 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`:

```
   Stack pointer
  └──*str
      └──str
          └──RET
              └──ebp
                  └──buf
```
Basic stack exploit

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

Stack pointer
Many unsafe C lib functions

\begin{itemize}
    \item \texttt{strcpy (char *dest, const char *src)}
    \item \texttt{strcat (char *dest, const char *src)}
    \item \texttt{gets (char *s)}
    \item \texttt{scanf (const char *format, \ldots)}
\end{itemize}

- "Safe" versions \texttt{strncpy()}, \texttt{strncat()} are misleading
  - \texttt{strncpy()} may leave buffer unterminated
  - \texttt{strncpy()}, \texttt{strncat()} encourage off by 1 bugs
    (overwrite LSB of SFP)
Exploiting buffer overflows

- Suppose web server calls `func()` with URL provided by the attacker
  - 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()` exits

- Sample remote buffer overflows of this type:
  - Picasa3 (1/2014)
  - RealNetworks RealPlayer (1/2014)
Control hijacking options

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

- **Function pointers:**
  - Overflowing `buf` will override function pointer `FunPtr`
  - e.g., PHP 4.0.2, MS MediaPlayer Bitmaps

- **Longjmp buffers**
  - `setjmp` saves registers (incl. SP, FP) to stack; `longjmp` restores registers
  - e.g., Perl 5.003
Other types of overflow attacks

- **Integer overflows**: (e.g., MS DirectX MIDI Lib)
  - Integers are represented using a fixed number of bits
  - Wrap around (modulo max value + 1) if value greater than max

[Graph showing the percentage of reported vulnerabilities that match “integer overflow” from 1996 to 2015.]

2. Integer overflow

```c
int catvars(char *buf1, char *buf2,
    unsigned int len1, unsigned int len2)
{
    char mybuf[256];
    if((len1 + len2) > 256)
        { /* [3] */ return -1; }
    memcpy(mybuf, buf1, len1); /* [4] */
    memcpy(mybuf + len1, buf2, len2);
    do_some_stuff(mybuf); return 0;
}

Problem: If len1 = 0x104 and len2 = 0xfffffffffc, then len1 + len2 = 0x100 (decimal 256), which allows buffer overflow attack!
```
3. Format string bugs

% of vulnerabilities that involve format string bugs

[National Vulnerability Database
History

- First exploit discovered in June 2000
- Examples:
  - `wu-ftpd 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`

Let’s try to understand the anatomy of this class of vulnerabilities
Format function execution

printf ("Number %d has no address, number %d has: %08x\n", i, a, &a);

From within the printf function the stack looks like this:

...<&a><a><i>A...

where:
A - address of the format string
i - value of the variable i
a - value of the variable a
&a - address of the variable a

Format function parses the format string ‘A’, by reading a character at a time

• If not ‘%’, the character is copied to the output

• If %, the character behind the ‘%’ specifies the type of parameter that should be evaluated; the parameter is located on the stack
Crashing a program: DoS attack

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

**Problem:** what if `user = "%s%s%s%s%s%s%s%s%s"`?
- Most likely program will crash
- Why?
  - `%s` will try to display memory from an address supplied on stack; very likely this is an illegal address, which is not mapped

**Correct form:**

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

... printf(user) ...

**Problem:** what if user =

```
"%08x.%08x.%08x.%08x\n"
```

Retrieve 4 parameters from stack and display them as 8-digit padded hexadecimal numbers
Buffer overflow using format string

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

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

**What if**

```c
user = "%500d \x3c\xd3\xff\xbf <nops> <shellcode>"?
```

- Bypasses "%400s" limitation
- `outbuf`: "Illegal command: [500-byte-long-number]\x3c..."
- Will overflow `outbuf`; now launch regular stack smashing buffer overflow attack

*Read team teso paper*
Summary: overflow exploits

- Understanding C functions and the stack
- Some familiarity with machine code
- Know how system calls are made

- 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 (Linux vs. Windows)
    - Stack growth direction

Read Phrack paper by Aleph One
Preventing hijacking attacks

- **Fix bugs:**
  - Audit software
    - Automated tools: Coverity, Prefast/Prefix.
  - Rewrite software in a type safe language (Java, ML)
    - C does not protect language abstractions
    - Rewriting difficult for existing (legacy) code …

- **Concede overflow, but prevent code execution**

- **Add runtime code to detect overflows exploits**
  - Halt process when overflow exploit detected
  - StackGuard, LibSafe, …

Will cover later

Today
Linux process memory layout

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

Memory Addresses:
- 0x08048000
- 0xC0000000
- 0x40000000
- 0x08048000
Marking memory as non-executable ($W^X$)

- 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)
- 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
Return to libc

- Control hijacking without code injection

```
stack
  args
  ret-addr
  sfp
  local buf

libc.so
  exec()
  printf()
  “/bin/sh”
```
Return to libc

- Control hijacking without code injection

### Perception:
- Attacker cannot execute arbitrary code
- Attacker relies on contents of libc — remove exec()?
The Return-oriented programming thesis

any sufficiently large program codebase

arbitrary attacker computation and behavior, *without* code injection

(Defense: control-flow integrity)

[Shacham 2007 for x86]
Return-oriented programming

- Treat libc as a corpus of instruction sequences each ending in “return” instruction
- Fill stack with pointers to these sequences (and with data)
- Execution flows through sequences, induces desired behavior (Turing complete)
- Falsifies perception of return-into-libc as limited, easy to defeat
Like rearranging magazine headlines …
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
    - Linux (via PaX): 16 bits of randomness for libraries
  - More effective on 64-bit architectures

- **Other randomization methods:**
  - Sys-call randomization: randomize sys-call IDs
  - Instruction Set Randomization (ISR)
ASLR example

Booting Vista twice loads libraries into different locations:

<table>
<thead>
<tr>
<th>Library</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></td>
<td>0x75370000</td>
<td>Windows NT MARTA provider</td>
</tr>
<tr>
<td>ntmarta.dll</td>
<td>0x6F2C0000</td>
<td>Shell extensions for sharing</td>
</tr>
<tr>
<td>ntshrui.dll</td>
<td>0x76160000</td>
<td>Microsoft OLE for Windows</td>
</tr>
<tr>
<td>ole32.dll</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Library</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></td>
<td>0x75660000</td>
<td>Windows NT MARTA provider</td>
</tr>
<tr>
<td>ntmarta.dll</td>
<td>0x6D9D0000</td>
<td>Shell extensions for sharing</td>
</tr>
<tr>
<td>ntshrui.dll</td>
<td>0x763C0000</td>
<td>Microsoft OLE for Windows</td>
</tr>
<tr>
<td>ole32.dll</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Run-time checking: StackGuard

- Many many run-time checking techniques ...
  - We’ll only mention a few

- **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
More methods …

- **StackShield**
  - At function prologue, copy return address `RET` and `ebp` to “safe” location (beginning of data segment)
  - Before return, restore saved `RET` and `ebp`
  - Implemented as assembler file processor (GCC)
Summary: Control hijacking attacks

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

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

- blexim, *Basic Integer Overflows*, Phrack 60, 2002

- Many of the slides are from D. Boneh and J. Mitchell’s Stanford CS 155 lecture on this topic
- Return-Oriented Programming slides based on slides from H. Shacham