CS 475: Lecture 3
Software Vulnerabilities

Rachel Greenstadt
April 14, 2015
Types of Software Vulnerabilities

• Databases : SQL Injection
• Web apps : XSS
• Broken crypto
• Buffer overflows (and related bugs)
• And more
History: Morris Worm

• Worm was released in 1988 by Robert Morris
  • Graduate student at Cornell, son of NSA chief scientist
  • Convicted under Computer Fraud and Abuse Act, sentenced to 3 years of probation and 400 hours of community service
  • Now an EECS professor at MIT (advised my Masters’ thesis)
• Worm was intended to propagate slowly and harmlessly measure the size of the Internet
• Due to a coding error, it created new copies as fast as it could and overloaded infected machines
• $10-100M worth of damage
Buffer Overflows and Morris Work

- One of the worm’s propagation techniques was a buffer overflow attack against a vulnerable version of `fingerd` on VAX systems
  - By sending special string to finger daemon, worm caused it to execute code creating a new worm copy
  - Unable to determine remote OS version, worm also attacked `fingerd` on Suns running BSD, causing them to crash (instead of spawning a new copy)
- CERT formed to deal with the new threat of software vulnerabilities
Buffer Overflows

• Common type of vulnerability
  • Often most common (depending on how you measure it)

• Tend to be critical as well
  • enable machine compromise
Memory buffer vulnerabilities

• Buffer is a data storage area inside computer memory (stack or heap)
  • Intended to hold pre-defined amount of data
    • If more data is stuffed into it, it spills into adjacent memory
  • If executable code is supplied as “data”, victim’s machine may be fooled into executing it – we’ll see how
    • Code will self-propagate or give attacker control over machine
• First generation exploits: stack smashing
• Second gen: heaps, function pointers, off-by-one
• Third generation: format strings and heap management structures
Software exploits/
Project 1

• Before you can understand stack exploits, you have to know something about computer architecture

• For project 1, you have to know x86 (IA-32)

• So we’ll do some review
Procedures

- Operating system runs programs as concurrently executed procedures
- The OS calls a program as a procedure to execute the program and the program returns control to the OS when it completes.
- The call to execute the procedure is a branch instruction to the beginning of the procedure. When the procedure finishes, a second branch instruction returns to the instruction immediately following the procedure call.
- The return address must be saved before the procedure is called. The steps in the transfer of control to execute a procedure are:
  1. Save the return address
  2. Call procedure (using a branch instruction).
  3. Execute the procedure.
  4. Return from the procedure (branch to the return address).
Nested procedure

- When the jal B instruction is executed, the return address in register $ra for procedure A will be overwritten with the return address for procedure B. Procedure B will return correctly to A, but when procedure A executes the jr instruction, it will return again to the return address for B, which is the next instruction after jal B in procedure A. This puts procedure A in an infinite loop.

- To implement the linkage for nested procedures, the return address for each procedure must be saved somewhere other than register $ra. Note that the procedure call/return sequence is a LIFO process: the last procedure called is the first to return. A stack is the natural data structure for saving the return addresses for nested procedure calls.
System Stack

- The system stack provides a convenient mechanism for dynamically allocating storage for the various data associated with the execution of a procedure including:
  - parameters
  - saved registers
  - local variables
  - return address

- The system stack is located at the top of the user memory space and grows downward toward smaller memory addresses. Register $esp is the stack pointer to the system stack. It contains the address of the first empty location at the top of the stack.
Linux process memory layout

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

%esp
brk

0x08048000
0x40000000
0xC0000000
The frame pointer is stored in register $ebp, also called $fp. A stack frame consists of the memory on the stack between the frame pointer and the stack pointer.
IA-32 Registers

- **$esp : Stack Pointer (SP) :** points to the top of the stack (lowest mem addr)
  - Points to last used word in stack or next available word location on stack (implementation dependent)

- **$ebp : Frame Pointer (FP) :** points to fixed location within an activation record (stack frame)
  - If $ebp for some stack frame is stored at addr X then $eip for that frame is stored at addr X + 4
  - Used to reference local vars and parameters since the distance from those to the frame pointer will not change whereas the distance from those to the stack pointer will (as other functions are called and the stack pointer is decrem’d …)

- **$eip : instruction pointer (aka $ra)**
  - “The instruction pointer (EIP) register contains the offset in the current code segment for the next instruction to be executed.”
Calling procedures (IA-32)

- When CALL procedure p()
  - Push eip : the return address ($ra)
  - Push ebp : saves previous frame pointer
  - Copy sp into fp : ebp = esp
    - The new stack frame’s frame pointer will be the previous value of the stack pointer
  - Advance sp (esp) for allocations on stack (that is, decrement it)

- When LEAVE procedure p(),
  - This process is reversed
  - Load ebp into esp
  - Restore ebp from the stack
Interaction between EIP, EBP, ESP

- During CALL, value of eip register pushed onto stack
- Before RET, programmer should make sure that stack pointer (esp) is pointing to the eip on the stack; does this via:
  - Move contents of ebp into esp
  - Increment esp by 4
  - esp should now point to (contain addy of) eip
  - RET will load the value stored in esp into the eip
Linux process memory layout

- **user stack**
- **shared libraries**
- **run time heap**
- **unused**
What are buffer overflows?

- Suppose a web server contains a function:
  ```c
  void func(char *str) {
      char buf[128]; /* Allocate local buffer
      128 bytes reserved on stack */
      strcpy(buf, str); /* Copy argument into
      local buffer */
      do-something(buf);
  }
  ```

- When the function is invoked, a new frame with local variables is pushed onto the stack:
What if buffer is overstuffed?

- Memory pointed to by str is copied onto the stack

- ```
  void func(char *str) {
      char buf[128];
      strcpy(buf, str); /*strcpy does not check sizeof buf */
      do-something(buf);
  }
  ```

- If a string longer than 128 bytes is written into buf it will overwrite adjacent memory locations:

- **These are often the saved registers!**
void function(char *str) {
    char buffer[8];
    strcpy(buffer,str);
}

void main() {
    char large_string[256];
    int i;
    for( i = 0; i < 255; i++)
        large_string[i] = 'A';
    function(large_string);
}
## Buffer Overflows

```c
void function(char *str) {
    char buffer[8];
    strcpy(buffer,str);
}

void main() {
    char large_string[256];
    int i;
    for( i = 0; i < 255; i++)
        large_string[i] = 'A';
    function(large_string);
}
```
Buffer Overflows

```c
void function(char *str) {
    char buffer[8];
    strcpy(buffer, str);
}

void main() {
    char large_string[256];
    int i;
    for( i = 0; i < 255; i++)
        large_string[i] = 'A';
    function(large_string);
}
```
Buffer Overflows

```c
void function(char *str) {
    char buffer[8];
    strcpy(buffer,str); }

void main() {
    char large_string[256];
    int i;
    for( i = 0; i < 255; i++)
        large_string[i] = 'A';
    function(large_string);
}
```
Buffer Overflows

```c
void function(char *str) {
    char buffer[8];
    strcpy(buffer, str); }

void main() {
    char large_string[256];
    int i;
    for (i = 0; i < 255; i++)
        large_string[i] = 'A';
    function(large_string);
}
```
# Buffer Overflows

```c
*str
ret (main)
0x41414141 ← FP
0x41414141
0x41414141 ← SP

void function(char *str) {
    char buffer[8];
    strcpy(buffer, str); }

void main() {
    char large_string[256];
    int i;
    for (i = 0; i < 255; i++)
        large_string[i] = 'A';
    function(large_string);
}
```
Buffer Overflows

```c
*str
0x41414141
0x41414141 ← FP
0x41414141
0x41414141 ← SP

void function(char *str) {
    char buffer[8];
    strncpy(buffer, str); }

void main() {
    char large_string[256];
    int i;
    for (i = 0; i < 255; i++)
        large_string[i] = 'A';
    function(large_string);
}
```
Buffer Overflows

```c
void function(char *str) {
    char buffer[8];
    strcpy(buffer,str); }

void main() {
    char large_string[256];
    int i;
    for( i = 0; i < 255; i++)
       large_string[i] = 'A';
    function(large_string);
}
```
Buffer Overflows

void function(char *str) {
    char buffer[8];
    strcpy(buffer, str);
}

void main() {
    char large_string[256];
    int i;
    for (i = 0; i < 255; i++)
        large_string[i] = 'A';
    function(large_string);
}

IP → 0x41414141

Segmentation Fault
Executing Attack Code

• Suppose buffer contains attacker-created string

• For example, *str contains a string read from the network as input to network daemon

• When function exits, code in the buffer will be executed, giving attacker a shell

• Root shell if the victim program is setuid root
Exploiting a Real Program

• It’s “easy” to execute our attack when we have the source code

• What about when we don’t? How will we know what our return address should be?
How to find Shellcode

1. Guess
   - time consuming
   - being wrong by 1 byte will lead to segmentation fault or invalid instruction
How to find Shellcode

2. Pad shellcode with NOP’s then guess
   - we don’t need to be exactly on
   - much more efficient
Small Buffer Overflows

• If the buffer is smaller than our shellcode, we will overwrite the return address with instructions instead of the address of our code

• Solution: place shellcode in an environment variable then overflow the buffer with the address of this variable in memory

• Can make environment variable as large as you want

• Only works if you have access to environment variables
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

```vbscript
Set test = CreateObject("Symantec.SymVAFileQuery.1")
test.GetPrivateProfileString "file", [long string]
```
/* Allocate memory for the response, size is 1 byte
 * message type, plus 2 bytes payload length, plus
 * payload, plus padding
 */
buffer = OPENSSL_malloc(1 + 2 + payload + padding);
bp = buffer;

/* Enter response type, length and copy payload */
*bp++ = TLS1_HB_RESPONSE;
s2n(payload, bp);
memcpy(bp, pl, payload);
bp += payload;
/* Random padding */
RAND_pseudo_bytes(bp, padding);

r = dtls1_write_bytes(s, TLS1_RT_HEARTBEAT, buffer, 3 + payload + padding);
Off-by-one Overflow

- 1-byte overflow: can’t change RET, but can change pointer to previous stack frame
  - On little-endian architecture, make it point into buffer
  - RET for previous function will be read from buffer!
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
  - Other heap bugs seen in IE 2008 and adobe PDF zero days
Format string problem

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

Problem: what if user = "%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, …
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")`
Writing Stack with Format Strings

◆ %n format symbol tells printf to write the number of characters that have been printed

```c
... printf("Overflow this!\n", &myVar); ...
```

- Argument of printf is interpreted as destination address
- This writes 14 into myVar ("Overflow this!" has 14 characters)

◆ What if printf does **not** have an argument?

```c
... char buf[16]="Overflow this!\n";
    printf(buf); ...
```

- Stack location pointed to by printf’s internal stack pointer will be interpreted as address into which the number of characters will be written.