Midterm

- One week from today
- Covers everything so far…
  **up to, but not including this class**
  - Class material
  - Readings, etc on Web site
  - Ect

- You can bring one 8.5x11 sheet of paper
  - Do anything you want with this paper
  - Must be readable without magnification
Quick statement on cheating

- We’ve found people trying to contract for solutions to Project 1; **this is cheating**
- We report all such cases of cheating to the Academic Integrity Office
- If you become aware of any cheating please report it to me, to the TAs or to the AIO. We take this seriously.
- This is a particularly bad class to try and cheat in.
Where we left off

- Last class: Return-oriented programming
  - We’ll finish now

- Today:
  - heap overflows,
  - integer overflows,
  - format string errors,
  - race conditions,
  - misc
Advanced technique: Return-oriented Programming

- Malicious code assumption
  - If I can prevent malicious code from being introduced or executed, then I’m fine

- Assumption turns out to be wrong
  - Malicious code is a subset of malicious computation
  - Ret-to-libc attacks are very simple example
    - No malicious code executed!
  - Turns out it can be generalized....
Thought experiment

- Suppose you have a stack overflow but can only redirect control flow to existing code
  - You can still jump to any legitimate instruction
- What if you jump into the middle of some code and that code ends with a RET instruction?
  - Where does control flow go now?
    » The return address pointed to by the stack pointer
  - Who controls that value?
    » The attacker does (because they had an overflow)
- The stack pointer increments; repeat
Return-oriented Programming
(bleeding edge: Hovav Shacham)

- Treat existing “good” code as a library
  Look for all code snippets that end in a “return”
  They do some little thing, but they can be “linked” together

- Lots of these on x86, because instructions are variable length, yet can begin on any byte sequence

- Example:

```
81 c4 88 00 00 00 add $0x00000088, %esp
5f pop %edi
5d pop %ebp
c3 ret
```
```
00 5f 5d ad db addb %bl, 93 (%edi)
c3 ret
```
Return-oriented Programming
(bleeding edge: Hovav Shacham)

Stack pointer (ESP) determines which instruction sequence to fetch and execute.
Processor doesn’t automatically increment ESP.
- But the RET at end of each instruction sequence does.
Return-oriented Programming
(bleeding edge: Hovav Shacham)

- It turns out you can use these sequences to build a “virtual instruction set”
- Can execute arbitrary bad computation
  But never introduce new code
  Only ever executes those “good” instructions

- Can be largely automated
  Two students here built a compiler for this in 2008
Example: Attack against AVC Advantage voting machine

Can only run code from ROM
Code injection impossible

Heap overflows

- Stack isn’t the only vulnerable data structure
- Clearly possible to overflow heap as well
- But is this reliably “exploitable” in general?
  - Prevailing wisdom in 1999 was no

- Real answer, “Yes, frequently”
  - 1999, Matt Conover (www.w00w00.org)
  - 2000, Solar Designer
  - 2002, Halvar Flake (BlackHat Briefings), 2004, David Litchfield (BlackHat Briefings) -> Windows
Simple heap overflow

- Idea: corrupt code pointer in heap

- Example: function pointer on heap
  ```
  struct * foo {
    int (*func)();
  }
  
  f = malloc(sizeof(struct foo));
  ```

- If *f can be overwritten via overflow, then callers of f->func() can be diverted to shellcode
**PointerGuard (Cowan et al)**

- Idea: Ok to let adversary corrupt code pointers if they can’t control contents

- Compiler support to *encrypt* all ptrs in memory
  - Decrypted before dereference
  - Encrypted before stores

- Attacker can corrupt pointer, but can’t control address without knowing the pointer *key*

- Also effective against many return-to-libc attacks

- But, expensive to do in general
  - EncodePointer/DecodePointer on Windows (manual)
Generic heap overflow

- Key idea: heap data structures holds both data and metadata (where allocated chunks are)
- The metadata holds pointers
  - Linked lists typically (allocated chucks vs free list)
- Heap impl writes through those pointers

- If you overwrite heap data into pointers you can control both the address and value
Typical problem (simplified)

- Each allocated memory chunk has a header
  
<table>
<thead>
<tr>
<th>prev (ptr)</th>
<th>next (ptr)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Data</td>
</tr>
</tbody>
</table>

- Used to track allocated/free memory
- Removing a block (a) from a list
  
  a.prev->next = a.next;
  a.next->prev = a.prev;

- What if you overwrite data block?
- A free may overwrite arbitrary location w/ arbitrary data
Lots of opportunities here

- Many places where heap pointers get written through
  - Memory coalescing
  - Look-aside lists (fast lookups)
  - Allocating new free memory from the OS
  - Etc.

- Highly dependent on *particular heap implementation*
What to do? Examples from Windows

- XPSP2
  - Safe List Removal
    - Validate on free Entry->next->prev == Entry->prev->next == Entry
  - 8-bit heap cookie tested on each free (like stack cookies)
  - Parts of heap have meta data encrypted with random number

- Vista
  - Randomize chunk metadata itself (encoded with random key)
  - Integrity check on malloc too (not just free)
  - Randomize location of heap data structures
  - Encrypt heap-related function pointers
  - Terminate program if error detected (why not make this mandatory?)

- And more in each new generation…

For details Google: Marinescu’s “Windows Vista Heap Management Enhancements” at Black Hat Briefings
Integer errors

- Strings aren’t the only datatype we screw up…
- In 2001, Michael Zalewksi finds overflow vuln caused by integer mishandling in SSH
  - Whole new class of hurting…
- Key issue:
  - Computer integers are not Platonic integers
  - Programmers don’t understand how integer expressions may evaluate based on input
  - C/C++ do not enforce range limits on arithmetic
Classic example

```c
void *ConcatBytes(void *buf1, unsigned int len1,
                  char *buf2, unsigned int len2)
{
    void *buf = malloc(len1 + len2);
    if (buf == NULL) return;

    memcpy(buf, buf1, len1);
    memcpy(buf + len1, buf2, len2);
}
```

What if:
len1 == 0xFFFFFFFFE
len2 == 0x000000102

0x100 bytes allocated... not enough. Ooops.
C/C++ Integer review
(type representation)

- **Type size (n bits)**
  - char (8 bits on x86)
  - short (16)
  - int (32)
  - long long (64)

- **Type signedness (signed or unsigned)**

- **Type range**
  - Signed: $-2^{n-1}$ to $2^{n-1}-1$ (2s complement machines)
  - Unsigned: $2^n - 1$
Kinds of errors

- Overflow
  - Occurs when integer increased above its maximum value or decreased below minimum value (sometimes called underflow)

- Truncation

- Sign conversion
**Truncation errors**

- Integer converted (via assignment) to one of smaller type and value cannot be contained
  - High-order bits lost; low order bits preserved
  - Tricky because of automatic type promotion in C

```c
char a, b, c;
a = 75;
b = 75
c = a + b;
```

150 > +127 (limit of unsigned char)

Expression promoted to int (C rules)

Truncation: -106
What if c was an index?
Sign errors

- Unsigned value converted to signed value of same length (no truncation)
  - Representation is the same
  - High-order bit interpreted as sign

```c
unsigned short a = 32768;
short b;

b = a;  // b = -32768
```

```c
a = 65535;
b = a;  // b = -1;
```
What to do?

- Strongly typed language
  - Most such problems go away in Java for example

- Runtime checking
  - gcc –ftrapv (trap on signed overflow on add, sub, mult)
  - Safe libraries (David LeBlanc’s SafeInt class)
    - Template overrides all operators (not the fastest)
    - Use when variable can be influenced by untrusted input

- Static checking (range analysis)
  - Sarkar et al, “Flow-insensitive Static Analysis for Detecting Integer Anomalies in Programs”
Using SafeInt

```c
void *ConcatBytes(void *buf1,
                  SafeInt<int> len1,
                  char *buf2,
                  SafeInt<int> len2)
{
    void *buf = malloc(len1 + len2);
    if (buf == NULL) return;
    memcpy(buf, buf1, len1);
    memcpy(buf + len1, buf2, len2);
}
```

Overload "+" will throw exception on overflow.
Format strings

- C supports *variable length* arguments
- These are used particularly for supporting printing and reading of formatted text (e.g. printf)
  - These functions typically have two parts:
    - Format string: text plus format specifiers
    - Arguments: manually matched to format specifiers
  ```c
  printf("%s %d\n", "10 plus 10 is", 20);
  ``
  ->
  "10 plus 10 is 20"

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Output</th>
<th>Passed as</th>
</tr>
</thead>
<tbody>
<tr>
<td>%d</td>
<td>decimal (int)</td>
<td>value</td>
</tr>
<tr>
<td>%u</td>
<td>unsigned decimal (unsigned int)</td>
<td>value</td>
</tr>
<tr>
<td>%x</td>
<td>hexadecimal (unsigned int)</td>
<td>value</td>
</tr>
<tr>
<td>%s</td>
<td>string ((const) (unsigned) char *)</td>
<td>reference</td>
</tr>
<tr>
<td>%n</td>
<td>number of bytes written so far, (* int)</td>
<td>reference</td>
</tr>
</tbody>
</table>
How is this implemented?

- **Caller**
  - Pushes arguments onto stack
  - Pushes pointer to format string onto stack

- **Callee**
  - Reads format string off stack
  - *Uses* format string to read arguments off of stack
    - Reads one value off stack for each “%” parameter
    - To be clear: printf runtime is looking up the stack, **controlled by % parameters**
Printf on the stack

f() {
    printf("%d\n", 10);
}

"%d\n"

10

Ptrn to string

return address

base pointer

Printf’s frame
Key problem

- User is responsible for enforcing one-to-one mapping between format specifiers and arguments

- What if there are too many arguments?

- What if there are too few arguments?
Exploiting format strings

Which is dangerous?

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

```c
int func(char *input) {
    fprintf(stdout, "%s", input);
}
```

Problem: what if input = "%s%s%s%s%s%s%s%s" ??

- Most likely program will crash. Why?
- If not, program will print memory contents.
Format Strings: reading

- Reading stack
  - `printf("%08x.%08x.%08x.%08x\n");`
  - prints four words off stack in hext

- What if we want to view other parts of memory?
  - Tricky: the format string itself is on the stack
  - Use "%08x" to move printf’s argument pointer back to string itself
  - `printf("x10\x01\x48\x08_%08x.%08x.%08x.%08x.|%s| ")`
    » Prints string at 0x08480110
    » Note: “x10\x01\x48\x08” = 0x08480110
    » Note: this assumes no intervening locals and four words of control data on stack before string pointer.
  - Can grovel in arbitrary memory for passwords, stack cookies, etc…

- But what about control flow?
Format Strings: overflow

- Format strings have a width modifier: min length

```c
{
    char      outbound[512];
    char      buffer[512];

    sprintf (buffer, "ERR Wrong command: %400s", user);
    sprintf (outbound, buffer);
}
```

- "\x501d\x3c\xd3\xff\xbf\x<nops><code>"
  - "\%501d" creates a string 501 characters long
  - With "Err Wrong Command: " is 8 bytes > buffer
  - Puts 0xbfffd33c into return address

- Pretty (un)lucky…
Format Strings: writing

- Special format specifier: “%n”
  - *Writes number* of bytes already printed
  - `printf( “hello %n”, &temp)`  -- *writes* ‘6’ into temp

- Use with reading trick to write to arbitrary memory
  - `printf(“\x10\x01\x48\x08 _%08x.%08x.%08x.%08x.%n”)`
    - Writes small integer into 0x08480110
  - By carefully constructing strings and writing each byte of a word separately can write arbitrary data to arbitrary places

For more detail read: Exploiting Format String Vulnerabilities by Team Teso
What to do?

- Remove the %n feature
  - Can’t: real programs use it

- Permit only static format strings
  - Can’t: real programs legitimately create dynamic format strings, especially for internationalization

- Match % directives to arguments at runtime
  - FormatGuard (Cowan et al)… match arguments to literals
    » Requires recompilation

- Analysis tools to detect format string vulnerabilities in source
  - Integrate format strings into type system (Wagner et al)

- Normalization: strip % from inputs
Misc

- Canonicalization
- Loop issues
- Bitfields
- Delete[]
Cannonicalization

- Sometimes there are multiple ways to represent the same data
  - “ ” is the same as “%20”
  - /etc/passwd is the same as /usr/homes/savage/../../../etc/passwd

- Classic problem
  - Validate input length on ASCII buffer
    - Yay! Will fit within allocated buffer
  - At later point, system coverts string to Unicode representation (oops, uses wide characters two bytes per symbol… overflow)
  - Famous Unicode overflow: CodeRed worm
  - Jizillions since…
The “Turkish-İ problem”  
(Applies also to Azerbaijan)

- Turkish has four letter ‘I’s
  - i (U+0069) I (U+0049) i (U+0131) İ (U+0130)
- In Turkish locale \( \text{UC("file")} == \text{FILE} \)

// Do not allow "FILE://" URLs
if(url.ToUpper().Left(4) == "FILE")
    return ERROR;
getStuff(url);

// Only allow "HTTP://" URLs
if(url.ToUpper(CultureInfo.InvariantCulture).Left(4) == "HTTP")
    getStuff(url);
else
    return ERROR;
Loop termination

char buff1[MAX_SIZE], buff2[MAX_SIZE];
out = buff1;
// make sure it’s a valid URL and will fit
if (!isValid(url)) return;
if (strlen(url) > MAX_SIZE - 1) return;
// copy up to first separator
do {
    // skip spaces
    if (*url != ' ') *out++ = *url;
} while (*url++ != '/');
strcpy(buff2, buff1);

... what if there is no ‘/’ in the URL?

Courtesy Jon Pincus
char buff1[MAX_SIZE], buff2[MAX_SIZE];
out = buff1;
// make sure it's a valid URL and will fit
if (! isValid(url)) return;
if (strlen(url) > MAX_SIZE - 1) return;
// copy up to first separator
do {
    // skip spaces
    if (*url != ' ') *out++ = *url;
} while (*url++ != '/') && (*url != 0);
strcpy(buff2, buff1);

1) order of tests is wrong
   what about 0-length URLs?
2) buff1 is not 0-terminated
Bitfields

- C/C++ allow bit-level data types
  ```
  struct {
    unsigned int a:8 (8 bits)
  } b;
  ```
  - Typically used to map onto bit-level file/stream formats
  - Vagueness in the standard leads to problems

- Truncation
  - Not clear how to handle bitfield as an rvalue (c = b.a)
    - gcc model: use length of type (i.e., int = 32 bits)
    - MSVC model: use length of bitfield (i.e. 8 bits)

- Sign conversion
  - What is type of b.a?
  - Not defined by standard, but many implementations implement it as a signed number!

- Bottom line: trivial to get this wrong
delete and delete[]

- Arrays of objects allocated/deallocated with new[] and delete[] in C++; not new and delete
- Incorrect code:
  ```c
  int main(void) {
      basebob *ba = (basebob *) new bob[4];
      dostuff(ba);
      delete ba;
  }
  ```
- Minor issue: only destructor for ba[0] is called
- Bigger problem: different heap representation

![Diagram showing heapChunk](image.png)

Courtesy Mark Dowd,
More unintuitive interactions

```c
BOOL DoStuff() {
    char pPwd[64];
    size_t cchPwd = sizeof(pPwd) / sizeof(pPwd[0]);
    BOOL fOK = false;
    if (GetPassword(pPwd, &cchPwd))
        fOK = DoSecretStuff(pPwd, cchPwd);
    memset(pPwd, 0, sizeof(pPwd));
    return fOK;
}
```

When DoStuff() returns can you still find the password on the stack?

Yes, compiler optimizes call to memset away…

Courtesy Mike Howard
TOCTOU vulnerabilities
Time of check/Time of use

- Key issue: program makes assumptions about *atomicity* of actions

```c
f() {
    check_something();
    then_do_something();
}
```

Is the thing you checked still true?
TOCTOU example

Scenario: root process wants to create a unique /tmp file

**Step 1:** choose a name

**Step 2:** check to see if it exists

**Step 3:** if it doesn’t exist, create it
Scenario: root process wants to create a unique /tmp file

**Step 1:** choose a name

**Step 2:** check to see if it exists

**Step 3:** if it doesn’t exist, create it

Here’s the problem:
- Attacker interrupts between steps 2 and 3
TOCTOU example

Scenario: root process wants to create a unique /tmp file

**Step 1:** choose a name
**Step 2:** check to see if it exists
**Step 3:** if it doesn’t exist, create it

Here’s the problem:
- Attacker interrupts between steps 2 and 3
- Creates a link (aka ln –s) from expected /tmp file name to a major file, i.e. /etc/passwd
TOCTOU example

Scenario: root process wants to create a unique /tmp file

Step 1: choose a name
Step 2: check to see if it exists
Step 3: if it doesn’t exist, create

Here’s the problem:

- Attacker interrupts between steps 2 and 3
- Creates a link from expected /tmp file name to a major file, i.e. /etc/passwd
- When program does the create, it stomps /etc/passwd with program’s authority
Quick background on suid

Unix programs have access rights one of which can be “setuid” +s

This means that the program can operate with the privileges of the program, not the caller

Use by programs like passwd (runs as root)

These programs need to be careful to check that the caller is allowed to do certain things
TOCTOU Example #2

- Code running with root/admin permissions
  /* access returns 0 if calling process could legally write to this file */
  if(!access(file, W_OK)) {
    f = fopen(file, "wb+");
    write_to_file(f);
  } else {
    fprintf(stderr, "Permission denied trying to open %s.\n", file);
  }

- Attack
  touch dummy; ln –s dummy pointer
  [call program with file=pointer; run again after access is called]
  rm pointer; ln –s /etc/passwd pointer
Is this realistic?

- You can’t control exactly when you run can you?

- Not usually, but you can try again and again and again… you only have to get it right once
What to do?

- Prevent or delay certain operations to the same filename in a given time (i.e. `stat()` followed by `link()`)

- Limit interleaving of operations to files from different processes (i.e. until one hasn’t touched file for x milliseconds)
Prevention tips

- Try to only operate on file descriptors, not names (immutable)
  - Especially avoid unlink()

- Don’t use access() to check privilege
  - Instead run program *unprivileged* and let file system do the checks for you
  - Operating system syscalls are usually atomic wrt the caller
This is just the surface...

- If you’re into this stuff,
  - Read Kotler’s “Advanced Buffer Overflow Methods” for more shellcode hacks
    » E.g. using program literals as serendipitous instructions; jumping into middle of instructions, etc
  - Read Dowd et al’s “Art of Software Security Assessment” for more nasty C/C++ issues (they also update a blog with new ones)
  - Vulnerability research literature on-line is very good (see next slide)… very dedicated people… I no longer believe in unexploitable bugs
If you’re interested in vulnerabilities...

- The important vulnerability research literature is generally not from academia

- Two classics
  - Smashing the Stack for Fun and Profit, AlephOne
  - The Tao of Windows Buffer Overflows, DilDog

- To keep up to date
  - Dave Aitel (Daily Dave mailing list)
  - H.D. Moore (browserfun.blogspot.com & metasploit)
  - Halvar Flake (ADD/XOR/ROL)
Next time...

- We’ll start Web vulnerabilities
  - Cross site scripting and SQL injection