Meta-questions

- Who are the authors?
- Why are they writing the paper?
When is a program secure?

- When it does exactly what it should?
  - Not more.
  - Not less.

- But how do we know what a program is supposed to do?
  - Somebody tells us? (But do we trust them?)
  - We write the code ourselves? (But what fraction of the software you use have you written?)
When is a program secure?

- 2nd try: A program is secure when it doesn’t allow **bad things**
- Easier to specify a list of “bad” things:
  - Delete or corrupt important files
  - Crash my system
  - Send my password over the Internet
  - Send threatening e-mail to the professor
- But… what if most of the time the program doesn’t do bad things, but occasionally it does? Or could? Is it secure?
What's a software vulnerability?

- A bug in a software program that allows an unprivileged user capabilities that should be denied to them

- Most general (i.e., worst)
  - Control flow hijacking
    - Divert control flow (in instruction stream)
    - Divert to “payload” that executes code of adversary’s choosing
Classic: Stack overflows

- Robert T. Morris worm, 1988
  (note: not control data)

- Cannon
  - AlephOne “Hacking the Stack for Fun and Profit”, Phrack 49, 1996
  - Dildog, “The Tao of Windows Buffer Overruns”, Cult of The Dead Cow cDC-351, 1998
  - Overwrite control data on stack to execute arbitrary instructions from input
Recap: Stack activations for C

Frame N

Stack Grown Down

Frame N-1

- Parameters
- Return Address
- Frame Pointer
- Locals
- Callee-save regs

- Parameters
- Return Address
- Frame Pointer
- Locals
- Callee-save regs
Example

```c
f() {
    g(parameter);
}

h(char *string) {
    char buf[16];
    strcpy(buf, string);
}
```

![Frame diagram]

- Parameters
- Return Address
- Frame Pointer
- Locals
- Callee-save regs

f()'s frame

g()'s frame

buf
What this looks like
(Windows x86 cdecl call)

Prolog

```assembly
push ebp       // save old frame pointer
mov ebp,esp    // Set current frame pointer
sub esp,10h    // reserve 16 bytes for buf
push ebx       // callee saves
push esi
push edi

...  do stuff

pop edi       // restore callee saves
pop esi
pop ebx
mov esp,ebp    // unroll stack
pop ebp        // restore old frame pointer
ret 3          // pop eip and jmp to it
```

Epilog

Caveat: no opt, no /GZ, no /GS
Quintessential stack overflow

- Basic problem is that the library routines look like this:

```c
void strcopy(char *src, char *dst) {
    int i = 0;
    while (src[i] != '\0') {
        dst[i] = src[i];
        i = i + 1;
    }
}
```

- If the memory allocated to dst is smaller than the memory needed to store the contents of src, a buffer overflow occurs.

- Particularly problematic with c’ s idiom of using local temporary buffers – allows “stack smashing” attack.
Stack smashing in action

```c
f() {
    g(badstring);
}

void g(char *string) {
    char buf[16];
    strcpy(buf, string);
}
```

- Parameters
- Return Address
- Frame Pointer
- Locals
- Callee-save regs

Evil shellcode address

Shellcode

16 bytes
Aside: why is it called shellcode?

```c
char shellcode[] = "\x31\xc0\x50\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62 \x69\x89\xe3\x50\x53\x50\xb0\x3b\xcd\x80";
```

Shellcode courtesy Foster, Osipov, Bhalla and Heinen
Early thinking

- Problem is that the string functions don’t have range checking

- Use versions that do have range checking and all will be well
  - e.g., `strncpy(char *dst, const char *src, size_t n)`
  - No more than n characters copied from *src to *dst
  - Simple no?
Problem: You have to use it right

- Vulnerability in htpasswd.c in Apache 1.3
  ```c
  strcpy(record, user);
  strcat(record, ":");
  strcat(record, cpw);
  ```

- “Solution”
  ```c
  strncpy(record, user, MAX_STRING_LEN-1);
  strcat(record, ":");
  strcat(record, cpw), MAX_STRING_LEN-1);
  ```

- Can write up to 2*(MAX_STRING_LEN-1) + 1 bytes!
What are the key issues?

- The language is weakly typed
  - Allows writing arbitrary values to arbitrary locations

- Control flow is dynamic, based on memory
  - Return address, function pointers, jump tables
  - If you overwrite these you can change control flow

- The processor doesn’t know the difference between code and data
  - It will execute instructions from any location in memory
Vulnerabilities, threats and hindsight

- Just a bug or exploitable vulnerability?

- Lots of hot air expended on this topic
  - “Yes, you found a bug, but it’s not exploitable”
  - “This class of bugs is very hard to exploit”
  - “While the DoS threat is significant, this vulnerability can’t be used for code injection”

- Historically these distinctions have changed with experience
  - Case in point: the off-by-one stack overflow
  - Historically, not considered a major control hijacking threat
  - Today, considered easy
```c
main() {
    f();
}

f() {
    g(input);
}

g(char *input) {
    char buf[16];
    int i;
    for (i=0;i<16;i++)
        buf[i]=input[i];
}
```

*Can overflow buffer by 1 byte!*

*When f returns control hijacked*

*Can overflow buffer by 1 byte!*

*Function epilog*

*...*

*mov esp,ebp // unroll stack*

*...*
Integer overflow: Classic example

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 == 0xFFFFFFFF
len2 == 0x000000102

0x100 bytes allocated... not enough. Ooops.

Courtesy Jon Pincus
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

150 > +127
(limit of unsigned char)

c = a + b;
```

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

a = 65535;
b = a; // b = -1;
```

Examples courtesy Robert Seacord
Aside: vulnerability research is "trendy"

- Example
  - Integer overflow reports from National Vulnerability Database
  - Zalewski identifies Integer overflow in OpenSSH in March of 2001
    - One more found 4 months later (tcpdump)

- Common pattern
  - Once new “class” of vulnerability is identified, then it gets found everywhere

- XP SP2 impact
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>Data</td>
<td></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?
- Write arbitrary data to arbitrary location
Language ambiguity: 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
Language ambiguity: delete and delete[]

- Arrays of objects allocated/deallocate with `new[]` and `delete[]` in C++; not `new` and `delete`
- Incorrect code:
  ```c
  int main(void) {
      basebob *ba = (basebob *) new bob[1024];
      dostuff(ba);
      delete ba;
  }
  ```
- Minor issue: only destructor for `ba[0]` is called
- Bigger problem: different heap representation

![Diagram showing malloc chunk header and variable allocation and deallocation](Image)

Courtesy Mark Dowd,
Attacks and Defenses

- What are the essential elements of control flow hijacking?

- What could you do to defend against or mitigate it?

- What could you do to go around those things?
Kinds of defenses

- Eliminate violation of runtime model
  » Better languages, code analysis

- Don’t allow bad input
  » Input validation

- Detect overflow/overwrite of data structures
  » Stack validation
  » Run-time bounds checking, pointer validation, etc
  » Reference monitors

- Don’t allow untrusted code to execute
  » Hardware protection, code signing

- Minimize invariants for making repeatable exploits
  » ASLR, code randomization, encrypted pointers

- Minimize impact of untrusted code running
Defense/attack pattern

- All attacks exploit programming assumptions that are not guaranteed
  - E.g., that integers don’t overflow, that inputs won’t exceed the size of the allocated buffer, etc.

- Defenses frequently do the same thing
  - Assume attacker requires X to mount attack and then try to prevent them from getting X
  - But what if they don’t really need X

- Security literature is full of attacks on defenses
Stack validation

- StackGuard, /GS, Propolice, etc
  - Insert secret “cookie” between locals and return address
  - Validate in function epilog
- Visual Studio 2002

```
Addl’ Prolog
{
sub    esp,24h
mov    eax,dword ptr _seccookie
mov    dword ptr [esp+20h],eax
}

Addl’ Epilog
{
mov    ecx,dword ptr [esp+20h]
xor    ecx,esp
add    esp,24h
jmp    _check_cookie
```
Memory protection

- Use hardware to prevent instruction fetch from certain pages
  - Modern RISC: clear execute bit in page table entries (PTEs)
  - Intel/AMD: NX extension (Intel called XD)

- Idea: mark stack pages as non-executable
  - When processor tries to execute injected code in stack it traps
  - Fault handler stops program
  - Called Data Execution Prevention (DEP) by Microsoft
Address Randomization

- Key idea: randomize code/data layout to minimize exploitable invariants
  - Example: randomize address of stack, heap
  - Assumption: attacker doesn’t know where to transfer control to
  - In Linux (PaX), MacOS and W7/Vista (ALSR)

- Issue: how difficult to determine address?
  - Some limitations on where objects can be located
  - Leak addresses through other interfaces
  - Hard to re-randomize once a program is running (e.g., server programs that fork())
Other kinds of low-level software attacks

» Return-to-libc
  ■ Chained function calling
  ■ Return-oriented programming

» Don’t know buf’s address
  ■ Trampolining (don’t know buff’s address)
  ■ NOP sleds

» Other kinds of overwrites
  ■ Function pointer clobbering
  ■ Data pointer overwrite (4 byte with/that)
  ■ Vtables, exception handlers
  ■ Format string
  ■ Heap overflow, heap spray
  ■ Type conversions

» Multi-stage attacks
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
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)
More stuff you could be reading...

- The important vulnerability research literature is generally **not** from academia

- To keep up to date
  - Dave Aitel (Daily Dave mailing list)
  - H.D. Moore (browserfun.blogspot.com & metasploit)
  - Halvar Flake (ADD/XOR/ROL)
  - Blackhat Briefings talks and some of the other cons (Shmoo, HITB, etc)
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
My favorite unintuitive interaction

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));
    *(volatile char*)pPwd = *(volatile char *)pPwd;
    return fOK;
}

Prevent optimization. Volatile tells compiler ptr can be changed/accessed outside program scope

Courtesy Mike Howard
For next time...

- Look at two kinds of defense papers
  - CFI—low-level control flow isolation
    (hugely influential, but still an open question)
  - Nozzle – system for detecting heap spraying attacks