CSE 127
Computer Security
Fall 2012

Software Security
Implementation Vulnerabilities I: Stack

Stefan Savage
Project #1

Timing Sidechannel Exercise

- On web pages

Due Nov 6th
When is a program secure?

What does that even mean?
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 do **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?
When is a program secure?

Claim: Perfect security rarely exists

- Security vulnerabilities are the result of violating an assumption about the software (or, more generally the entire system)
- Corollary: As long as you make assumptions, you’re vulnerable.
- And: You always need to make assumptions! (or else your software is useless and slow)

However, some assumptions are more dangerous than others
What is a software vulnerability?

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

Most problematic kind?

- **Control flow hijacking**
- Bug that allows input data to be *executed as code*

Buffer overruns are the quintessential example
Buffer overflows

The #1 source of serious vulnerabilities in software

Caused because C and C++ are not safe languages

For example:

- They use a “null” terminated string representation: “HELLO!\0”
- Standard library routines *assume* that strings will have the null character at the end (vs Pascal)
- Bad defaults: the library routines don’t check inputs

Easy to accidentally get wrong
...even easier to maliciously attack
**Buffer overflow attacks**

Assumption (by programmer) is that the data will fit in a limited buffer

This leads to a vulnerability: Supply data that is too big for the buffer... violating assumption

This vulnerability can be exploited to subvert the entire programming model

- i.e. execute arbitrary code
What you need to do to hijack control of a system

Get your code into victim’s address space (placement)
Get their program to jump there (diversion)

Lets look at stack overflows first...
Recap: Stack activations for C

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

Stack Grown Down

Frame N

Frame N-1
Example

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

g(char *string) {
    char buf[16];
    strcpy(buf,string);
}
```
What this looks like
(Windows x86 cdecl call)

Prolog

{ 
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

Epilog

{ 
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

Caveat: no opt, no /GZ, no /GS
First problem: 

**strcpy() is unsafe**

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);
}
```

- **Evil shellcode address**
- **Shellcode**
- **24 bytes**

Diagram:
- Parameters
- Return Address
- Frame Pointer
- Locals
- Callee-save regs
Aside: why is it called shellcode?

```
xor    eax, eax
push   eax
push   0x68732f6e
push   0x69622f2f
mov    ebx, esp
push   eax
push   eax
push   ebx
mov    al, 59
push   eax
int    80h
```

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

//bin/sh

Shellcode courtesy Foster, Osipov, Bhalla and Heinen
Some issues

What textual restrictions on attack string?
- Can’t have a NULL

How do you know where the return address is on the stack?
- Deterministic relative to local buffer on stack frame

How do you know what the precise address of your shellcode is?
- Deterministic location: Trial and error (educated)
- Semi-deterministic (or lazy): “NOP sled’s” to the rescue (0x90)
Lots of string-based buffer overflows

Nov 2, 1988, fingerd (morris worm)
- Interestingly, not a control flow violation
- change stack-allocated string passed to exec() from “/usr/ucb/finger” to “/bin/sh”

July 17, 2000 Microsoft Outlook
July 26, 2000 Adobe Acrobat
May 18, 2000, Kerberos v4
… 1000’s
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!
char *copy(char *s) {
    char buffer[BUF_SIZE];
    strncpy(buffer, s, BUF_SIZE-1);
    buffer[BUF_SIZE-1]= '\0';
    return buffer;
}

This program returns a pointer to local memory.
More strncpy misuse... What’s wrong with this code?

```c
void main(int argc, char **argv) {
    char program_name[256];
    strncpy(program_name, argv[0], 256);
    f(program_name);
}
```

String program_name may not be null terminated.
Much bigger problem...
its not just strings

memcpy, bcopy
Arrays
Pointer arithmetic on local buffers
etc...
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

Bug or exploitable vulnerability?
- No threat
- Denial-of-service threat
- Arbitrary remote code execution threat

Lots of hot air expended on this topic
- “Yes, you found a bug, but its 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
Off-by-one example

```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

Fake frame with ret address

Parameters

Return Address

Frame Pointer

Locals

Callee-save regs
Nice overview of buffer overflows

http://nsfsecurity.pr.erau.edu/bom/

Courtesy Rick Ord
What to do?

Compile time approaches
- Memory-safe languages
- Testing tools

Run-time approaches
- Prevent control data
  - Stack validation (Stack cookies)
  - Memory protection (DEP)
- Protect jump target
  - Randomization (ASLR)
Compile-time

Memory-safe languages
- Java, C#, ML, Haskell, etc
- Type system prevents buffer overflow from being expressed
- Legacy code issues, performance, interfacing with raw I/O

Testing tools
- Fuzz testing (explicitly test for excessive inputs, off-by-one, integer overflow, etc with values likely to cause crash)
- Program analysis (PreFIX/PreFAST, Coverity, etc)
  » Detects presence of overflows, but doesn’t prove absence
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 
\}
How is cookie secret?

Picked randomly
Initialized at program startup
Different each time program is run

Assumption: adversary doesn’t know it
Stack validation

Lots of details to get right…
Early StackGuard had cookie before frame pointer…
consequence?
◆ What’s the fix?
Stack validation

Same issue with
- Exception handlers
- Function pointers

Need to put cookie between vulnerable buffer and any control flow data
Stack validation issues

Must recompile program
- New function prolog/epilog, stack layout
- Standard on Windows since XP SP2

Performance overhead
- Not so bad (2-3%) for stack

Doesn’t protect against all code hijacks
- Just overflows of local buffers into control data in the stack frame
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
Issues w/Memory protection

Can break some real programs
- JITs, Scheme/LISP, some self-modifying code tricks

Doesn’t prevent control flow hijack only limits target
- Heap-based shellcode
  - Store shellcode into malloced buffer
  - Redirect control flow there; trickier to protect heap
- Return-to-libc attacks
  - Overwrite ret addr to point to well known library function (e.g. system()) and pass appropriate arguments on stack (e.g. “/bin/sh”).
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())
Advanced techniques: Heap spray

Can cause control flow into heap, but don’t know where your shellcode is stored (ASLR)

Basic idea: overwhelming force

- Allocate jizzillions of copies of the shellcode (with big NOP sleds) and then jump blindly into the heap

Very common with today’s “drive-by download” attacks on vulnerability

- Heap spray implemented using Javascript
Drive-By Heap Spraying

Owned!
**Drive-By Heap Spraying (2)**

Program Heap

ASLR prevents the attack

bad

PC

- Creates the malicious object
- Triggers the jump

```html
<iframe src=file://BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB... NAME="CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC..."
width="100%"
height="100%"
frameborder="0"
allowfullscreen"
></iframe>
</html>
```
Drive-By Heap Spraying (3)

Program Heap

Allocate 1000s of malicious objects

```javascript
<SCRIPT language="text/javascript">
    shellcode = unescape('%u4343%u4343...'');
    oneblock = unescape('%u0C0C%u0C0C');
    fullblock = oneblock;
    while (fullblock.length<0x40000) {
        fullblock += fullblock;
    }
    sprayContainer = new Array();
    for (i=0; i<1000; i++) {
        sprayContainer[i] = fullblock + shellcode;
    }
</SCRIPT>
```
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
```
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
Next time...

More software vulnerabilities...

Heap vulnerabilities
Integer errors
Format string vulnerabilities
Misc...