

# Course Objectives

After completing this course, students will be able to:

* Summarize the CTE squad's responsibilities, objectives, and deliverables from each CPT stage
* Analyze threat information
* Develop a Threat Emulation Plan (TEP)
* Generate mitigative and preemptive recommendations for local defenders
* Develop mission reporting
* Conduct participative operations
* Conduct reconnaissance
* Analyze network logs for offensive and defensive measures 

# Course Objectives (Continued)

Students will also be able to:

* Analyze network traffic and tunneling protocols for offensive and defensive measures
* Plan non-participative operations using commonly used tools, techniques and procedures (TTPs)

# Module 2: Threat Emulation (Objectives)

* Conduct reconnaissance
* Generate mission reports from non-participative operations  Plan a non-participative operation using social engineering
* Plan a non-participative operation using Metasploit
* Analyze network logs for offensive and defensive measures
* Analyze network traffic and tunneling protocols for offensive and defensive measures
* Plan a non-participative operation using Python
* Develop fuzzing scripts
* Develop buffer overflow exploits

Module 2 — Lesson 1 3: Fuzzing (Objectives)

* Define fuzzing
* Explain the purpose of fuzzing
* Identify the components of fuzzing
* Perform manual fuzzing
* Develop fuzzing scripts
* Determine potential vulnerabilities
* Use a third-party fuzzer to test an application

# Lesson Overview

In this lesson we will discuss:

* Overflowing buffers
* Manually
* Simple Python scripts
* Fuzzers
* Developing your own fuzzer
* Create your own simple network fuzzer with Python
* Improve functionality and flexibility of your fuzzer
* Preparing to conduct buffer overflow exploits 

What is Fuzzing

* Automated testing
* "using malformed/semimalformed data injection"
* "random bad data"
* "see what breaks"

Owasp.org defines fuzzing as:

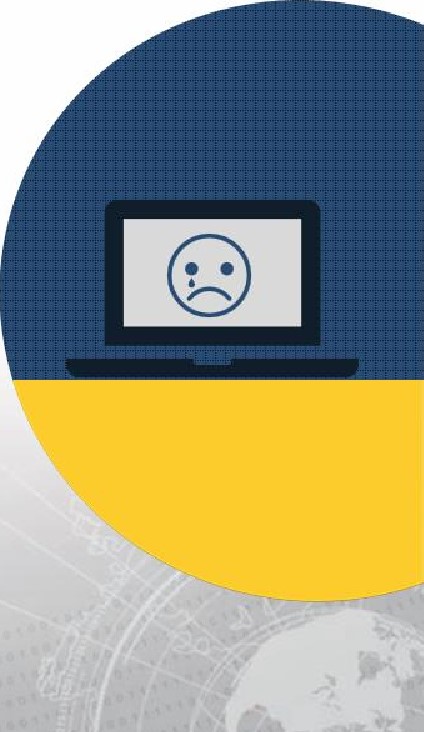
|  |
| --- |
|  |
| "...a Black Box software testing technique, which basically consists in finding implementation bugs using malformed/semi-malformed data injection in an automated fashion."  - https://www.owasp.org/index.php/Fuzzing |

IBM's Elliotte Rusty Harold defines fuzzing as:

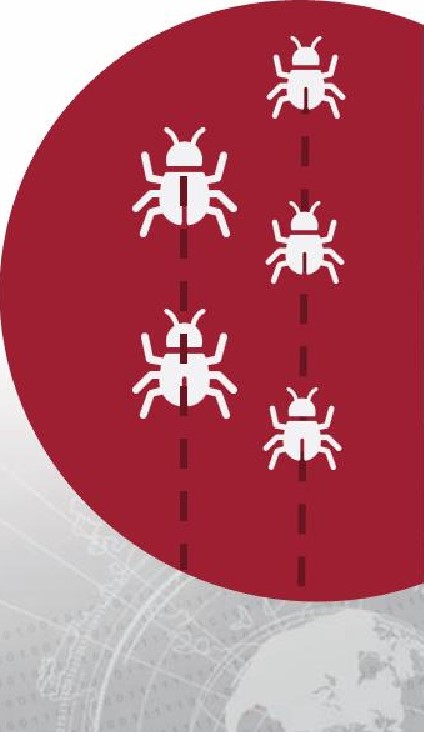
|  |
| --- |
|  |
| "In fuzz testing, you attack a program with random bad data (aka fuzz), then wait to see what breaks. The trick offuzz testing is that it isn't logical: Rather than attempting to guess what data is likely to provoke a crash (as a human tester might do), an automated fuzz test simply throws as much random gibberish at a program as possible. "  - https://www.ibm.com/developerworks/library/jfuzztest/index.html |

# Fuzzing Goals

Make stuff fall over...

* Systematically providing inputs with the goal of making bad things happen
* Looking for bad things:
* Crashes
* Hanging
* "Use-after-free" discovery
* Memory leaks
* Unexpected behavior

Why Fuzz?

* Black box testing
* Finds bugs in applications when you don't have the source code
* Most fuzzing is done in black box testing, but...
* White box testing
* Find bugs faster than just doing code review, cast wider net
* Automation
* Many aspects of fuzzing can be fully automated, freeing developers and analysts to focus on other aspects of the program/system

# Basic Fuzzing Vectors

* Various attack vectors: 145713445 • Text/characters 35373845101
* Numbers

HYOW}-HPLWQP

* Binary data
* Metadata I<ZJHDWBCMWE

# TJELPSIOLSW

## 10010101011

101101010

101010

Basic Fuzzing Vectors — Improved

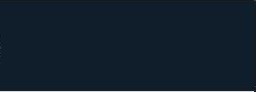
• Known dangerous vectors:

### • Integers: zero, negative numbers, and very large numbers

* Characters (chars): Escaped characters, interpretable (injection like attacks), odd symbols (É S?), or simply a large amount of characters
* Binary: random ones

### OWASP Fuzz Vectors

* Recursive vs Replacive

Recursive

* Sharing a trait with brute force, recursive fuzzing involves iterating through all possible combinations of a valid character or alphabet
* e.g. Input: aaaa, Input: bbbb, Input: cccc, etc.

Replacive

* Fuzzing by replacing a known input with new data. This new data could be pseudo random or it could be known attack vectors.  e.g. SQL injection, LDAP injection, etc.



Fuzzing

categories

#### Generation Fuzzing

* "Intelligent" or "smart" fuzzing
* Generates input from scratch
* Generates input based on user provided instructions
* Knowledge of the system/application/protocol

Mutation Fuzzing

* "Dumb fuzzers"
* Modifies existing (real) input
* Utilizes a corpus of seed input
* i.e. If the application inputs text, provide a large amount of valid text. If the application analyzes images, provide a library of valid images.

Other Types of Fuzzers

|  |
| --- |
| Context-free, Grammarbased Generation |

 Evolutionary 

* Monitors behavior • Generation-based and responses and fuzzer that utilizes generates test expressions, terms, cases based on operations, factors those observations and constants
* Arguably one of the most useful styles of fuzzing currently available



Taint-based Directed

* "Whitebox" fuzzing that is aware of possible vulnerable points and focuses testing at those points.

### Catching the Results

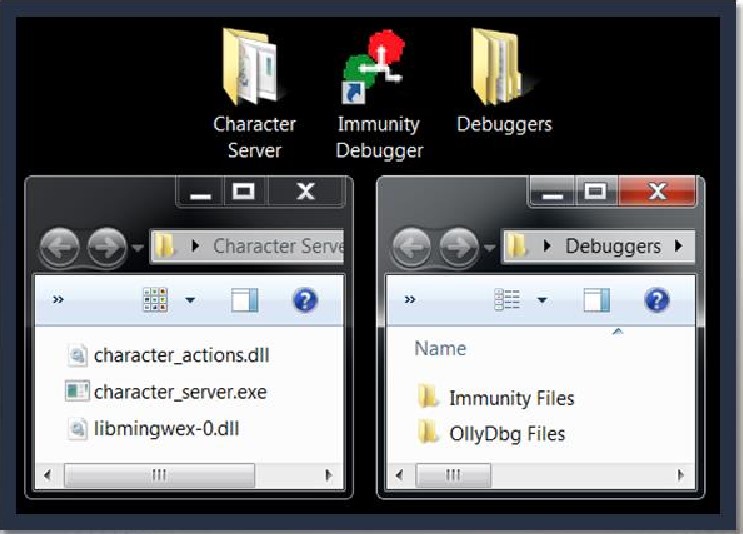
* Simply fuzzing isn't enough on its own, we need to monitor the u Its :
* Monitor for exceptions
* One of the most common results from fuzzing are crashes and other exceptions
* Utilize debuggers
* Can hook a process and help pinpoint where an exception occurred
* Monitor responses
* Various methods ranging from watching the screen to advanced scripts and even hardware devices

#### Let's find a buffer overflow: Initial Setup

* Open your Lesson 13 environment which will include:
* Windows 7
* Kali Linux
* Take note of the IP address for each
* Ping each machine from the other to verify connectivity and firewall status
* This is a network based example, so firewalls need to be off.

### Important Files and Directories

• On Windows 7 we will be working with a program called "Character Server" and debuggers, specifically Immunity (Optionally OllyDbg)

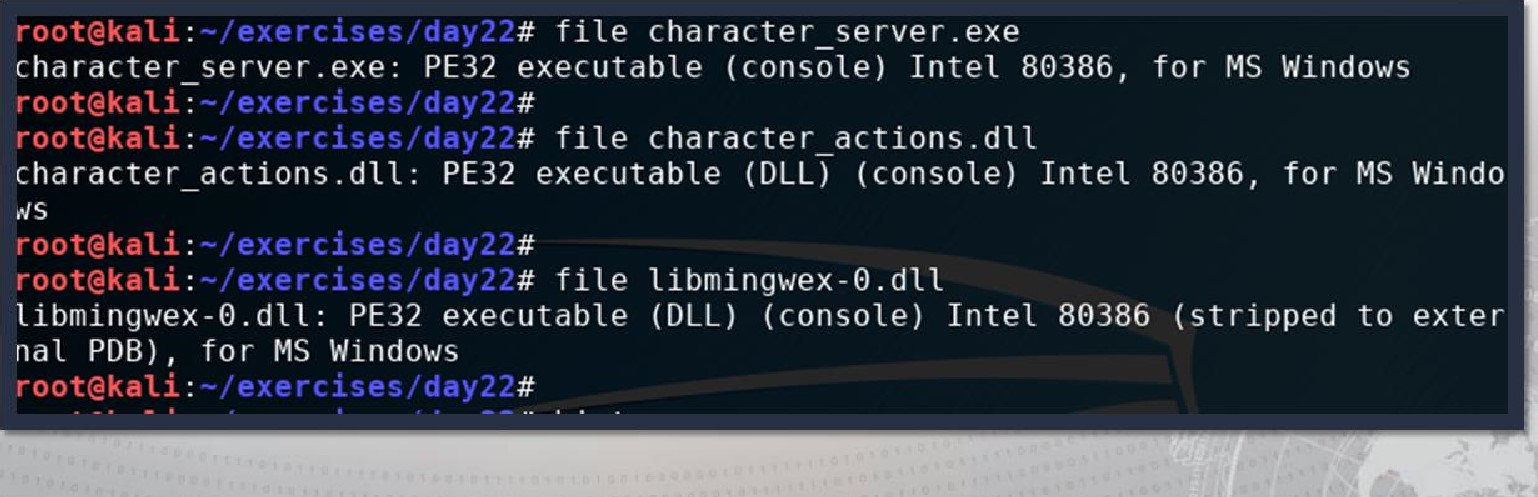


### "Character Server"

* SCENARIO: You've discovered that your target is using a piece of software called "Character Server." You have obtained a copy of the executables and DLL's but not the source code. You are not able to identify any open source information about this application, its uses, or its vulnerabilities.
* TASK: Identify vulnerabilities through simple fuzzing.

What type of file is this anyways?

* Character Server is made up of 1 EXE file and 2 DLL files according to the directory. But it is often prudent to check the header info to be sure,



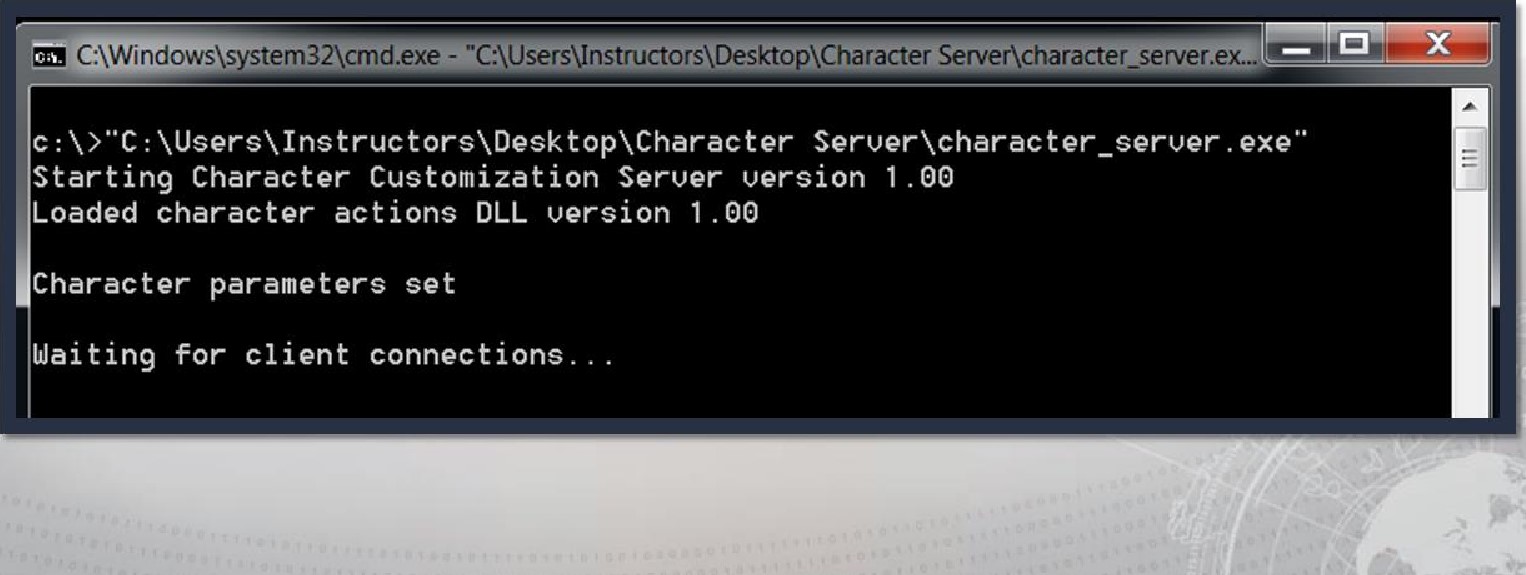
#### Setting up something to fuzz

* character server.exe:
* PE32 executable (console) Intel 80386, for MS Windows



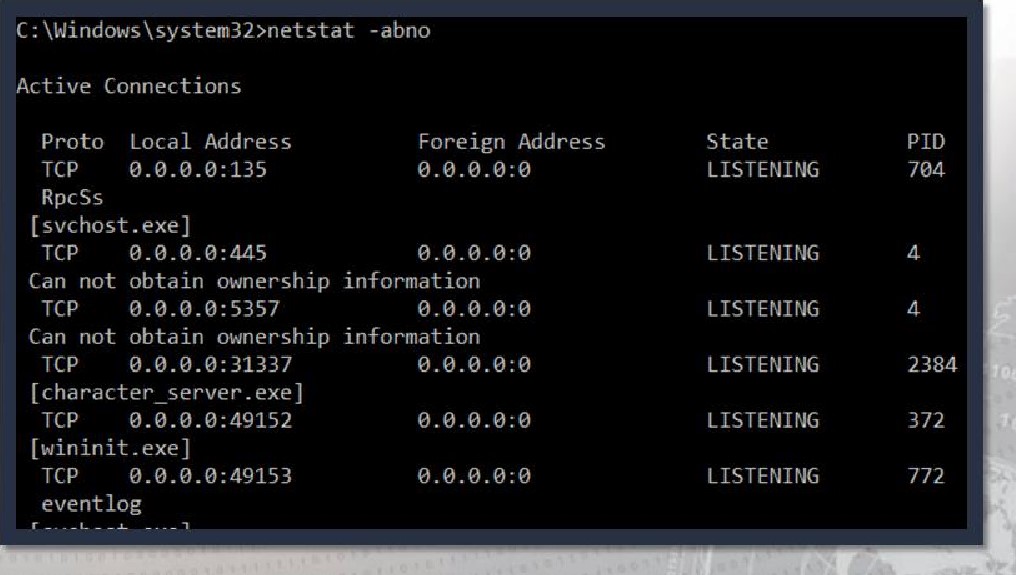
Running the server to be tested

* Because it is a Windows console program, let's run it on our Windows box



#### Determining server network behavior

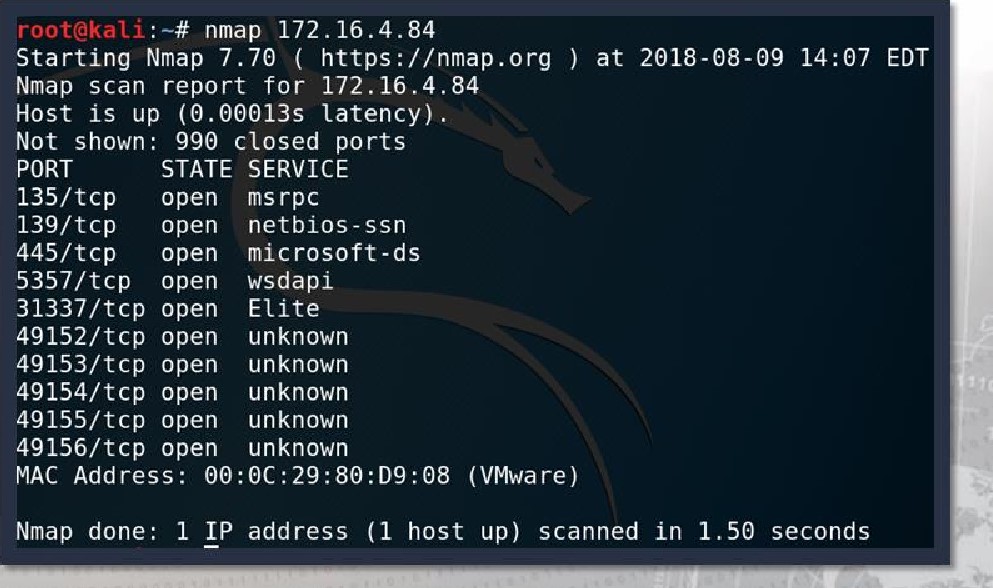
* Looks like the server is listening for connections on the network, let's see what port it is using.

ooct

What's happening on that port?

* Various tools to help you interrogate an open port

. Nmap

. Amap

* Netcat
* Many more...

Digging deeper with Amap

* Amap is an Application MAPper
* Deeper dive than some other tools

root@kali amap 172.16.4.84 31337 amap v5.4 (www.thc.org/thc-amap) started at 2018-08-09 - APPLICATION MAPPING mode

Unrecognized response from (by trigger http) received.

Please send this output and the name of the application to vh@thc . org:

5765 6c63 6f6d 6520 746f 2074 6865 2043 Welcome to the C

6861 7261 6374 6572 2043 7573 746f 6d69 haracter Customi

0020 : 7a61 7469 6f6e 2053 6572 7665 7221 2054 zation Server! T

0030 7970 6520 4845 4c50 2066 6f72 206f 7074 ype HELP for opt 696f 6e73 2eOa 554e 4b4e 4f57 4e20 434f ions. -UNKNOWN CO

4d4d 414e 44@a IMMAND .

Unidentified ports: 172.16. 4.84.•31337/tcp (total 1) .

amap v5.4 finished at 2018-08-09

#### Let's connect and find more

* Using netcat we can establish a raw connection to this port/service

root@kati . nc 172.16.4.84 31337

Welcome to the Character Customization Server! Type HELP for options.

* Pretty obvious here isn't it? Type HELP

HELP

FIRST

LAST NICK

RACE

CLASS

GROUP

ROLE

ROLL

SAVE

CALC

EXIT

[character\_first\_name]

[character\_last\_name]

[character nickname]

[human, elf, drow, orc]

[warrior, wizard, cleric]

[g roup\_options I

[character role]

[die number]

\_

[save file name]

[save file name]

It accepts input

* Seems at least one of these commands accepts text input:



Can we overflow the buffer?

* Let's throw some characters in there and see.. .

|  |
| --- |
|  |
| NICK blahblahblahblahblahblahblahblah    Nickname set to blahblahblahblahblahblahblahblah |
|  |

* Let's check the server and see if we have any errors. Nope, let's try more...

NICK blahblahblahblahblahblahblahblahblahblahblahblahblahblahblahblah

Nickname set to blahblahblahblahblahblahblahblahblahblahblahblahblahblahblahblah

* Okay, this is going to take a while.

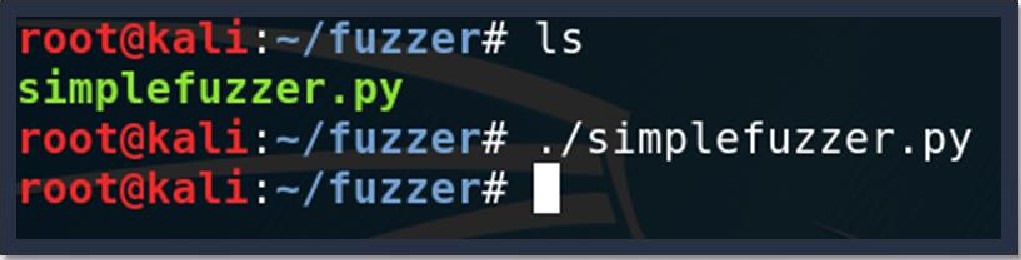
### Python to the rescue

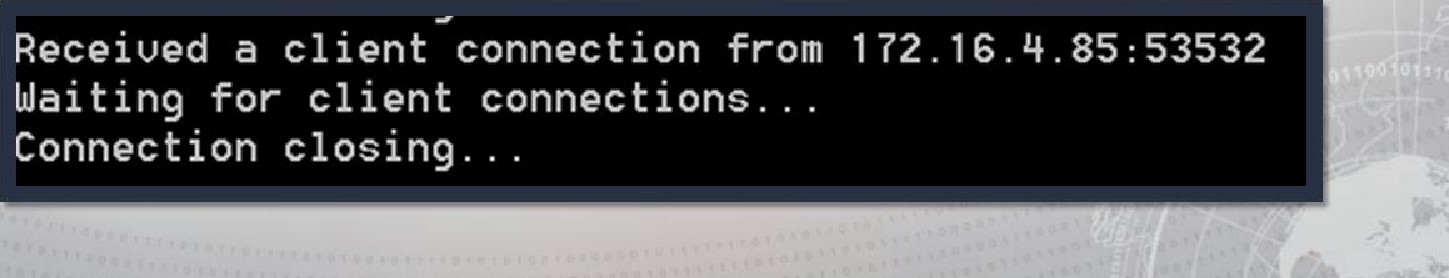
a• We have the option of typing "random" characters for days, but let's use Python to automate the task for us.



Does it work?

* Let's run it against the application and see if anything falls over.





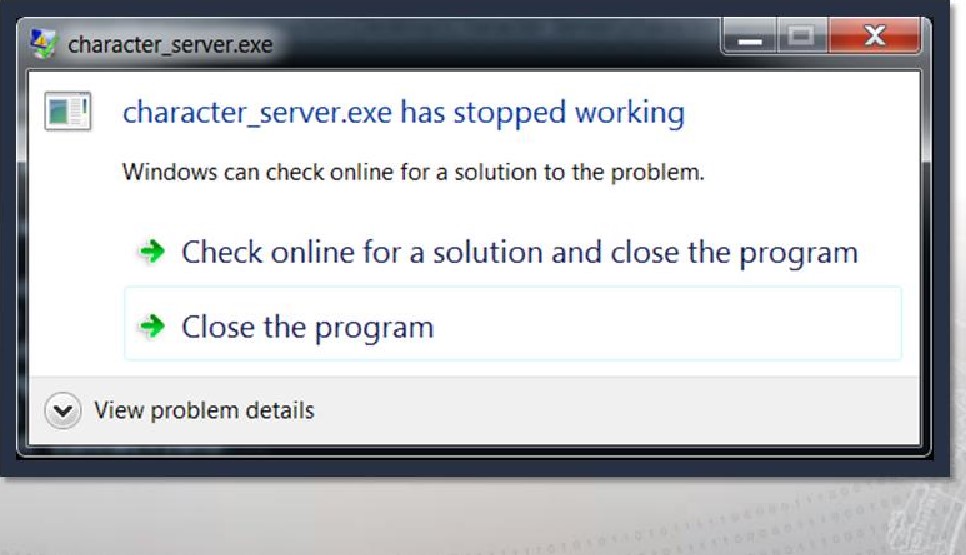
We can definitely do it this time

* Let's increase the buffer size slightly.

|  |  |  |
| --- | --- | --- |
| buf = | "NICK " + "A" | \* 100000000 |
|  |  |  |

### It fell over

• Looks like 100 million A's will work

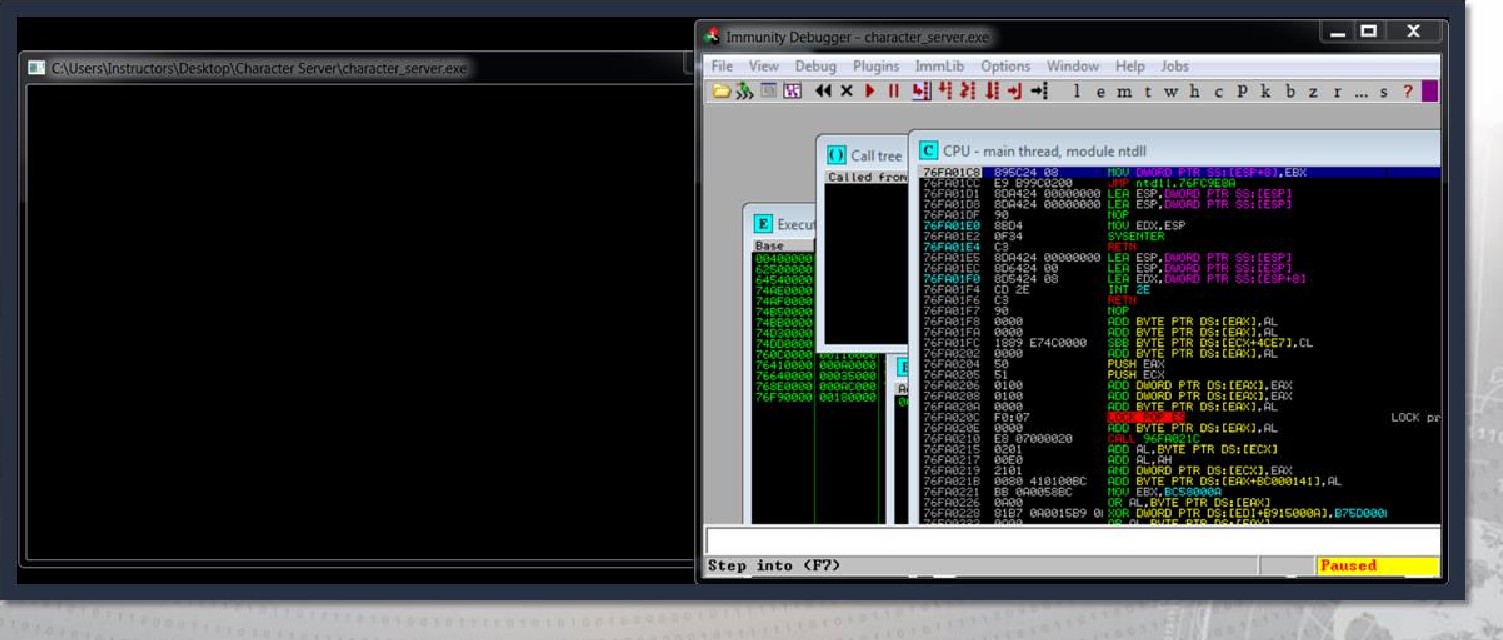


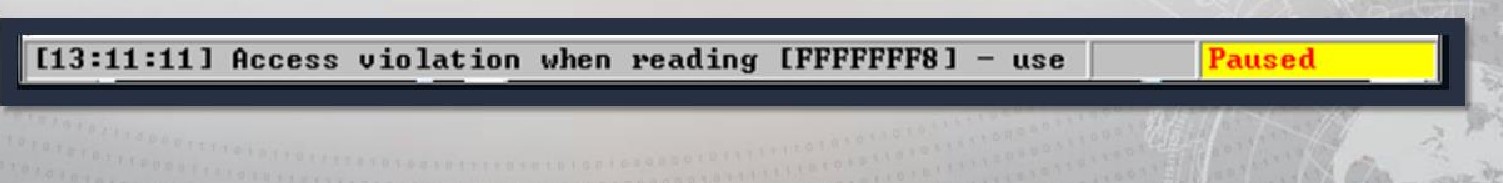
### Catching the results, a better way

* We've seen a crash in Windows, now let's try a debugger (Immunity)
* Multiple ways to hook a process
* Drag and drop executable onto debugger icon
* Open debugger, then File > Open
* Open debugger, launch application, then File > Attach

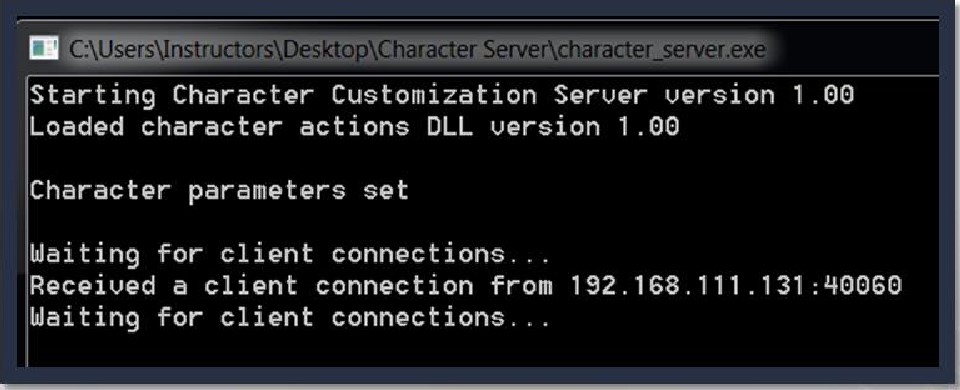
#### Attaching or opening the process

* We decided to open the executable in Immunity with File > Open



Now, let's try again with the debugger

* Run the script again, and check the results in Immunity



Why'd we just do that?

* Goal of fuzzing?
* Make something fall over
* Crashes are only the first step
* Monitoring results is critical
* What's next?

• Turn crashes into exploits!

### An example of 3rd party fuzzer

Sometimes, it's just easier/faster to use a 3rd party tool.

* SPIKE (one of those 3rd party fuzzers)
* Well known, poorly documented
* Scalable
* Allows custom C programming

SPIKE Commands

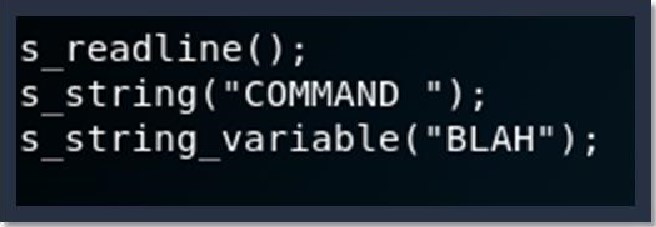
SPIKE provides a framework for customized fuzzing scripts and commands to execute them. Here are some of those commands:

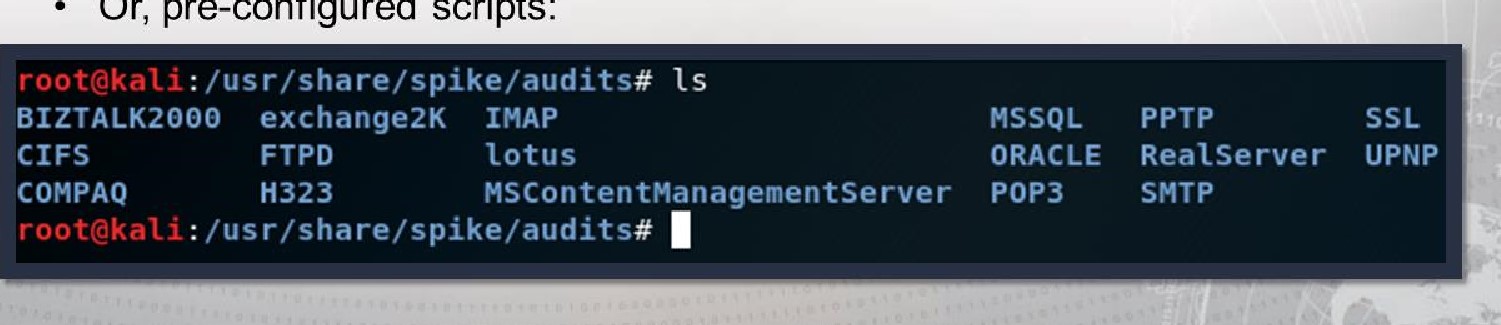
generic chunked generic send tcp generic web server fuzz generic listen tcp generic send udp generic web server fuzz2



## SPIKE Scripts

SPIKE scripts provide the framework for customized C code fuzzers.

• Example (simple) script: 



Or,

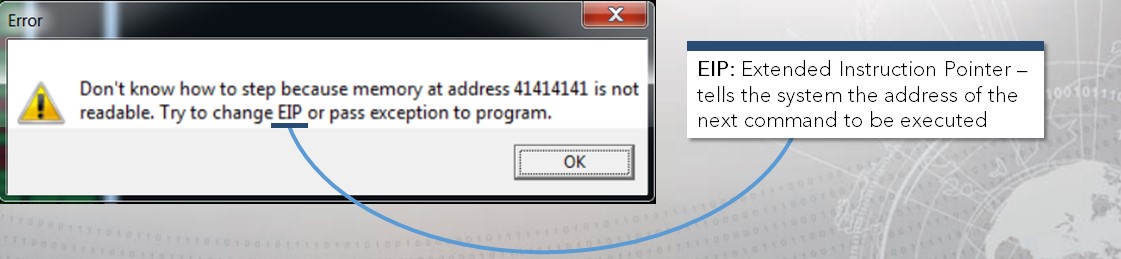
pre-configured

scripts:

### Capturing SPIKE Results

SPIKE is a very fast fuzzer that sends tons of data. So how do we know if it worked?

* Capturing the data:
* Attach target process to debugger
* Open Wireshark and configure it to monitor all fuzzing attempts
* Try to step through (F 7) and see if EIP is corrupted



Launching a script in SPIKE

Once a script has been configured, launching it is straightforward.

For our purposes we'll be utilizing generic\_send\_tcp:

### /usr/bin/generic\_send tcp <port> <script.spk> 0 0



## Exercise: Simple Fuzzer

Objectives

After completing this exercise, students will be able to:

* Perform manual fuzzing
* Develop fuzzing scripts
* Determine potential vulnerabilities

Duration

This exercise will take approximately 5 hours to complete.

### Debrief

General Questions

* How did you feel about this procedure?  Were there any areas in

particular where you had difficulty?

* Do you understand how this relates to the work you will be doing?

Specific Questions

. Were you able to manually fuzz" any of the commands?

* If yes, what techniques did you use?
* What increment did your fuzzer count by in the CHALLENGE?
* Which commands did you try?

## Exercise: SPIKE

Objectives

After completing this exercise, students will be able to:

• Use a third-party fuzzer to test an application

Duration

This exercise will take approximately 1 hours to complete.

### Debrief

General Questions

* How did you feel about this procedure?  Were there any areas in

particular where you had difficulty?

* Do you understand how this relates to the work you will be doing?

Specific Questions

* Did a crash occur?
* If yes, what techniques/scripts did you use?
* Did you capture the string that crashed the command?
* Whatwas the exact string?

## Exercise: Intermediate Fuzzer (BONUS)

Objectives

After completing this exercise, students will be able to:

* Perform manual fuzzing
* Develop fuzzing scripts
* Determine potential vulnerabilities

Duration

This exercise will take approximately 5 hours to complete.

### Debrief

Specific Questions

* Were you able to "manually fuzz" any of the commands?
* What increment did your fuzzer count by in the challenge?
* Did a crash occur?
* Did you capture the string that crashed the command?

Lesson Summary

In this lesson we learned about:

* Overflowing buffers
* Manually
* Simple Python scripts

. Fuzzers

* Developing your own fuzzer
* Create your own simple network fuzzer with Python
* Improve functionality and flexibility of your fuzzer
* Prep before conducting buffer overflow exploits

|  |
| --- |
| End of Module 2, Lesson  1 3 |