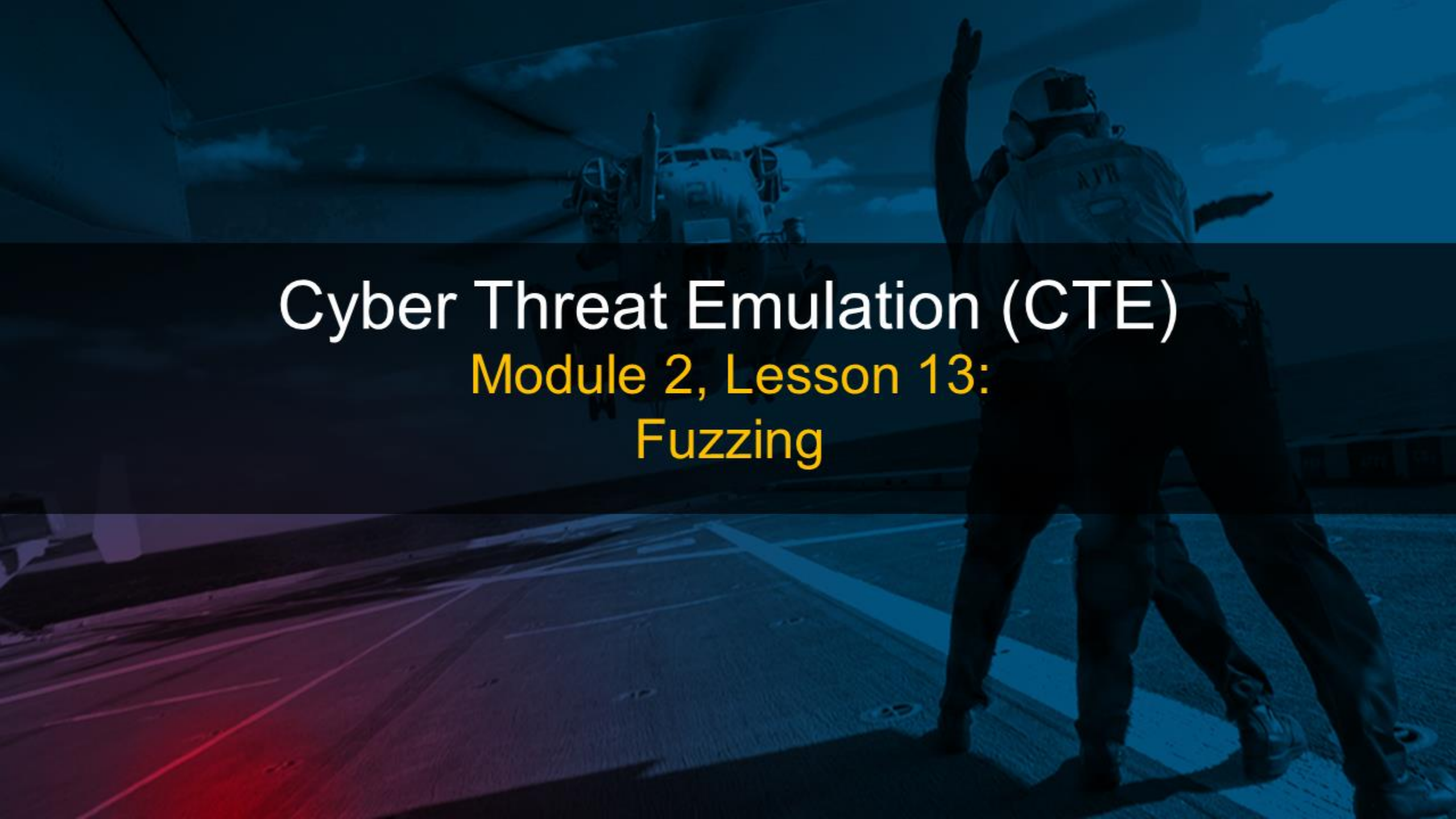


# Cyber Threat Emulation (CTE)

## Module 2, Lesson 13:
## Fuzzing

# 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 13: 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/semi-malformed 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 of fuzz 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/j-fuzztest/index.html

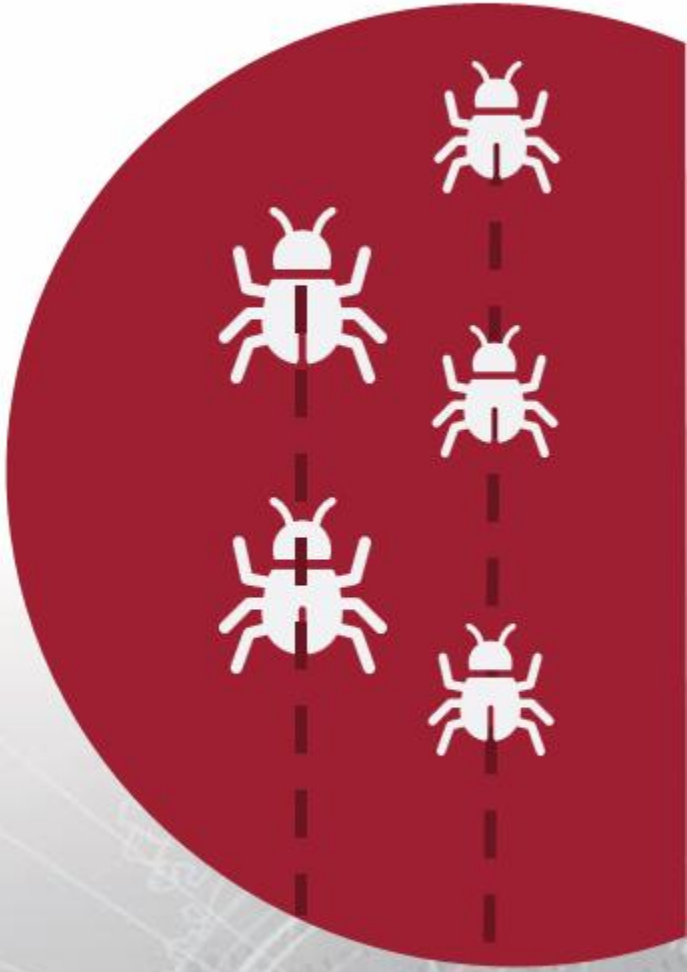
# Fuzzing Goals

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:
  - Text/characters
  - Numbers
  - Binary data
  - Metadata

# 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 (¢ ≈ š?), or simply a large amount of characters
  - Binary: random ones

# OWASP Fuzz Vectors

- Recursive vs Replacive

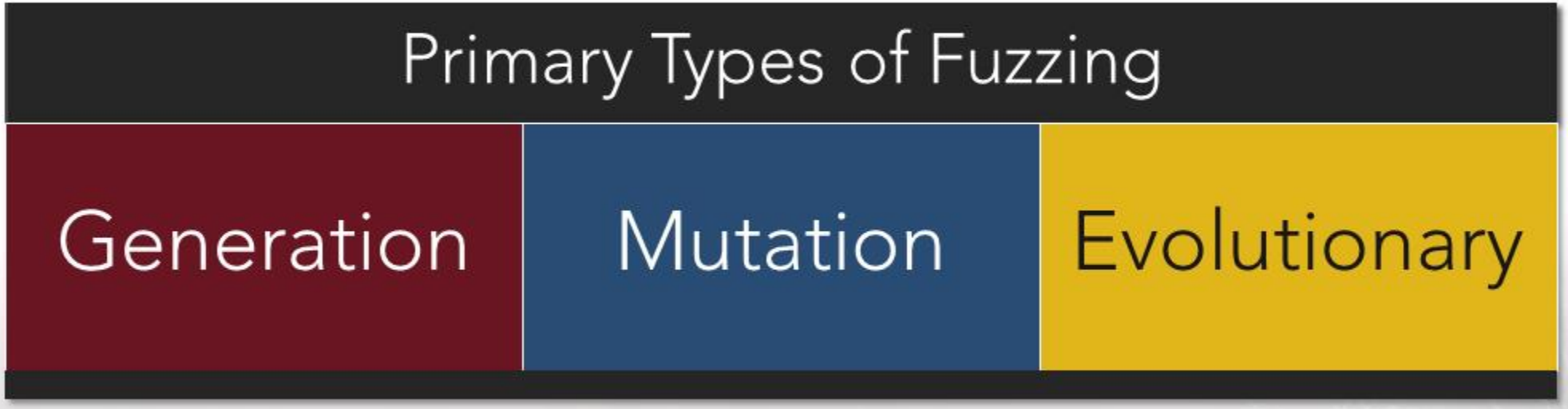
## Recursive

- 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

## Evolutionary

- Monitors behavior and responses and generates test cases based on those observations

## Context-free, Grammar-based Generation

- Generation-based fuzzer that utilizes expressions, terms, operations, factors 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

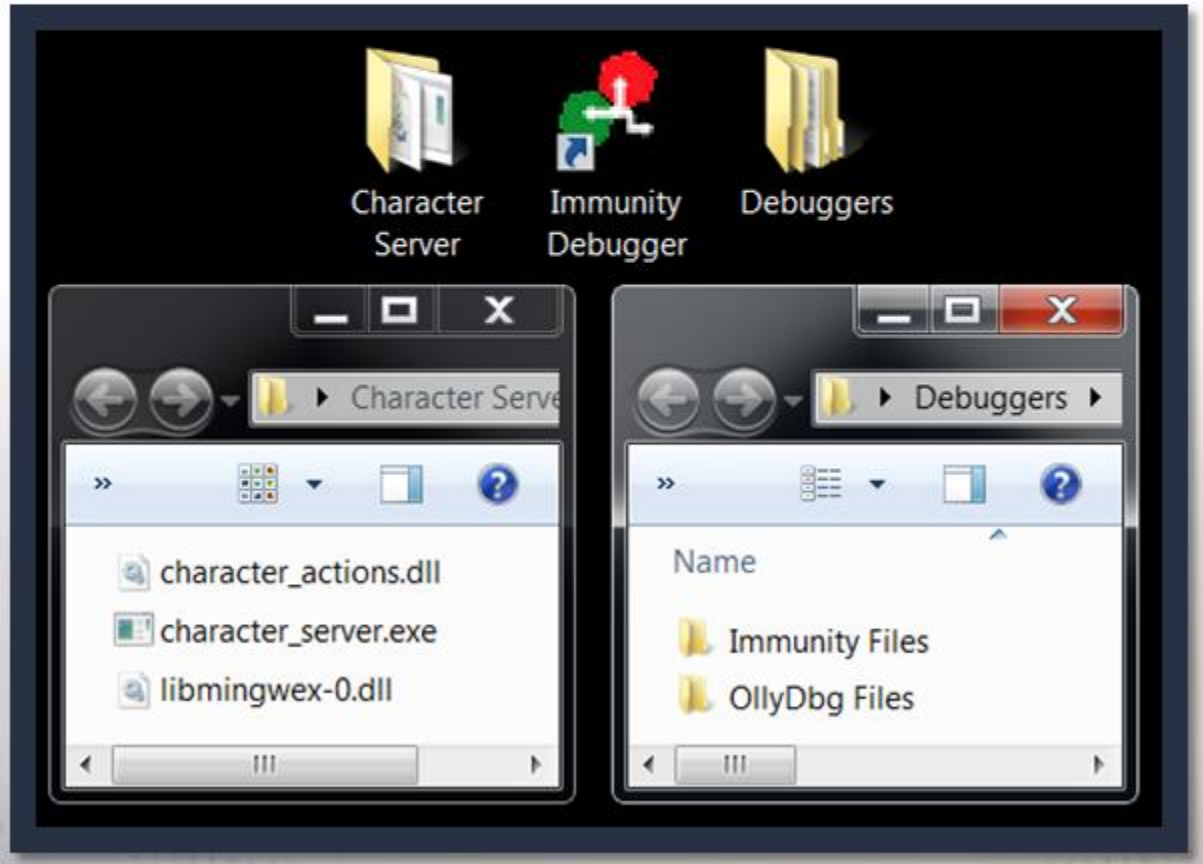
- Simply fuzzing isn't enough on its own, we need to monitor the results:
  - Monitor for exceptions
    - One of the most common results from fuzzing are crashes and other exceptions
  - Utilize debuggers
    - Can hook a process and help pinpoint where an exception occurred
  - Monitor responses
    - 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.

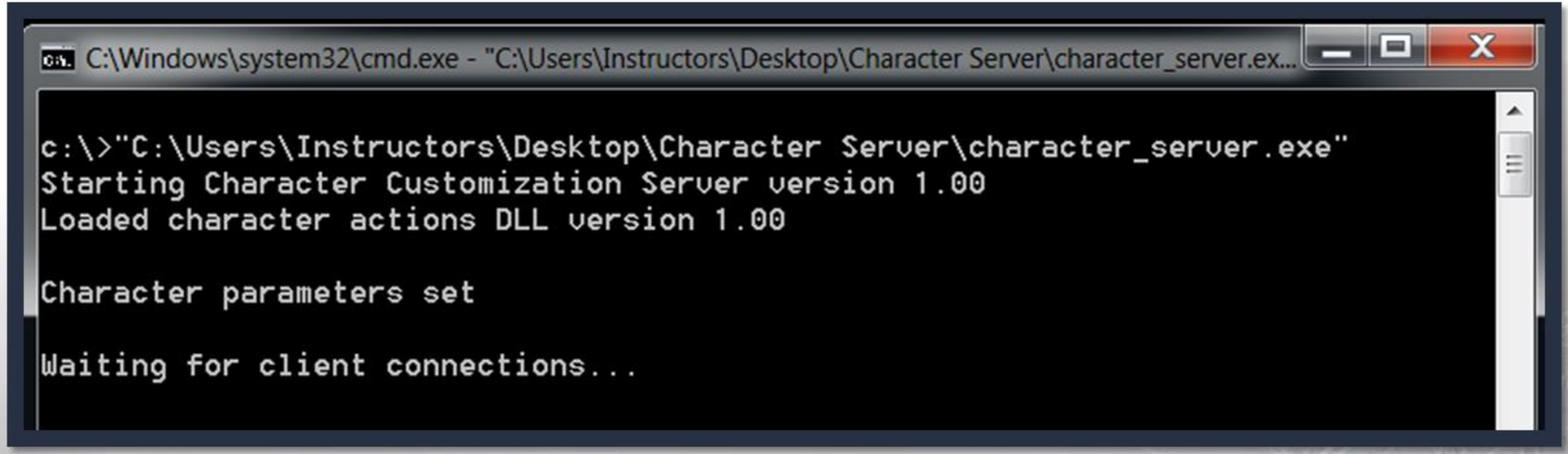
```
root@kali:~/exercises/day22# file character_server.exe
character_server.exe: PE32 executable (console) Intel 80386, for MS Windows
root@kali:~/exercises/day22#
root@kali:~/exercises/day22# file character_actions.dll
character_actions.dll: PE32 executable (DLL) (console) Intel 80386, for MS Windo
ws
root@kali:~/exercises/day22#
root@kali:~/exercises/day22# file libmingwex-0.dll
libmingwex-0.dll: PE32 executable (DLL) (console) Intel 80386 (stripped to exter
nal PDB), for MS Windows
root@kali:~/exercises/day22#
```

# 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.

```
C:\Windows\system32>netstat -abno

Active Connections

  Proto  Local Address          Foreign Address        State           PID
  TCP    0.0.0.0:135            0.0.0.0:0              LISTENING       704
  RpcSs
 [svchost.exe]
  TCP    0.0.0.0:445            0.0.0.0:0              LISTENING       4
 Can not obtain ownership information
  TCP    0.0.0.0:5357           0.0.0.0:0              LISTENING       4
 Can not obtain ownership information
  TCP    0.0.0.0:31337          0.0.0.0:0              LISTENING       2384
 [character_server.exe]
  TCP    0.0.0.0:49152          0.0.0.0:0              LISTENING       372
 [wininit.exe]
  TCP    0.0.0.0:49153          0.0.0.0:0              LISTENING       772
  eventlog
```

# What's happening on that port?

- Various tools to help you interrogate an open port
  - Nmap
  - Amap
  - Netcat
  - Many more…

```
root@kali:~# nmap 172.16.4.84
Starting Nmap 7.70 ( https://nmap.org ) at 2018-08-09 14:07 EDT
Nmap scan report for 172.16.4.84
Host is up (0.00013s latency).
Not shown: 990 closed ports
PORT      STATE SERVICE
135/tcp   open  msrpc
139/tcp   open  netbios-ssn
445/tcp   open  microsoft-ds
5357/tcp  open  wsdapi
31337/tcp open  Elite
49152/tcp open  unknown
49153/tcp open  unknown
49154/tcp open  unknown
49155/tcp open  unknown
49156/tcp open  unknown
MAC Address: 00:0C:29:80:D9:08 (VMware)

Nmap done: 1 IP address (1 host up) scanned in 1.50 seconds
```

# 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 14:09:50 - APPLICATION MAPPING mode

Unrecognized response from 172.16.4.84:31337/tcp (by trigger http) received.
Please send this output and the name of the application to vh@thc.org:
0000:  5765 6c63 6f6d 6520 746f 2074 6865 2043    [ Welcome to the C ]
0010:  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 ]
0040:  696f 6e73 2e0a 554e 4b4e 4f57 4e20 434f    [ ions..UNKNOWN CO ]
0050:  4d4d 414e 440a                             [ MMAND.           ]

Unidentified ports: 172.16.4.84:31337/tcp (total 1).

amap v5.4 finished at 2018-08-09 14:09:50
```

# Let's connect and find more

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

```
root@kali:~# 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 [character_first_name]
LAST  [character_last_name]
NICK  [character_nickname]
RACE  [human, elf, drow, orc]
CLASS [warrior, wizard, cleric]
GROUP [group_options]
ROLE  [character_role]
ROLL  [die_number]
SAVE  [save_file_name]
CALC  [save_file_name]
EXIT
```

# It accepts input

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

```
NICK Valicor
Nickname set to Valicor
```

# Can we overflow the buffer?

- Let's throw some characters in there and see…

```
NICK blahblahblahblahblahblahblahblah
Nickname set to blahblahblahblahblahblahblahblah
```

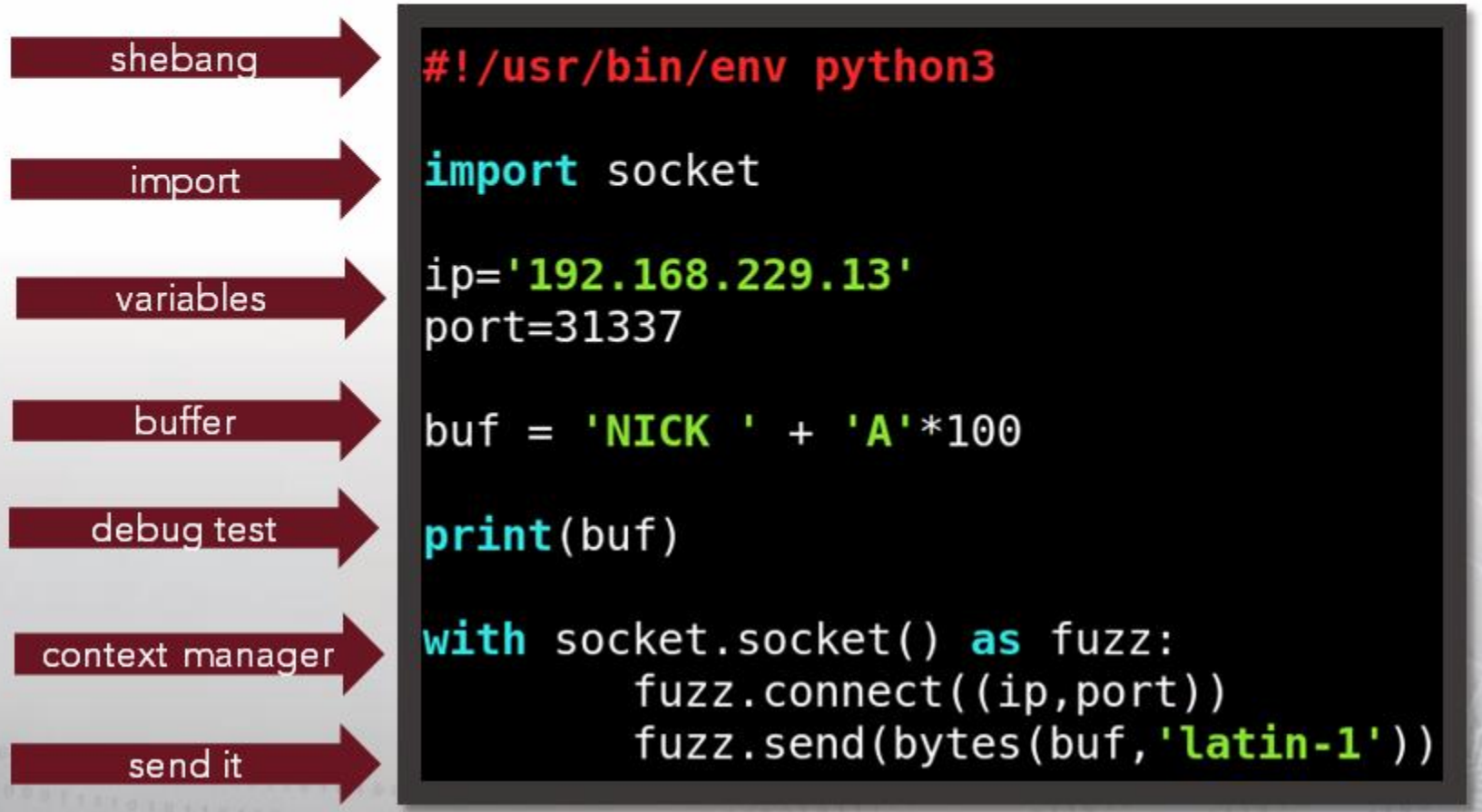
- Let's check the server and see if we have any errors.  Nope, let's try more…

```
NICK blahblahblahblahblahblahblahblahblahblahblahblahblahblahblahblah
Nickname set to blahblahblahblahblahblahblahblahblahblahblahblahblahblahblahblah
```

- Okay, this is going to take a while.

# Python to the rescue

- 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.

```
root@kali:~/fuzzer# ls
simplefuzzer.py
root@kali:~/fuzzer# ./simplefuzzer.py
root@kali:~/fuzzer#
```

```
Received a client connection from 172.16.4.85:53532
Waiting for client connections...
Connection closing...
```

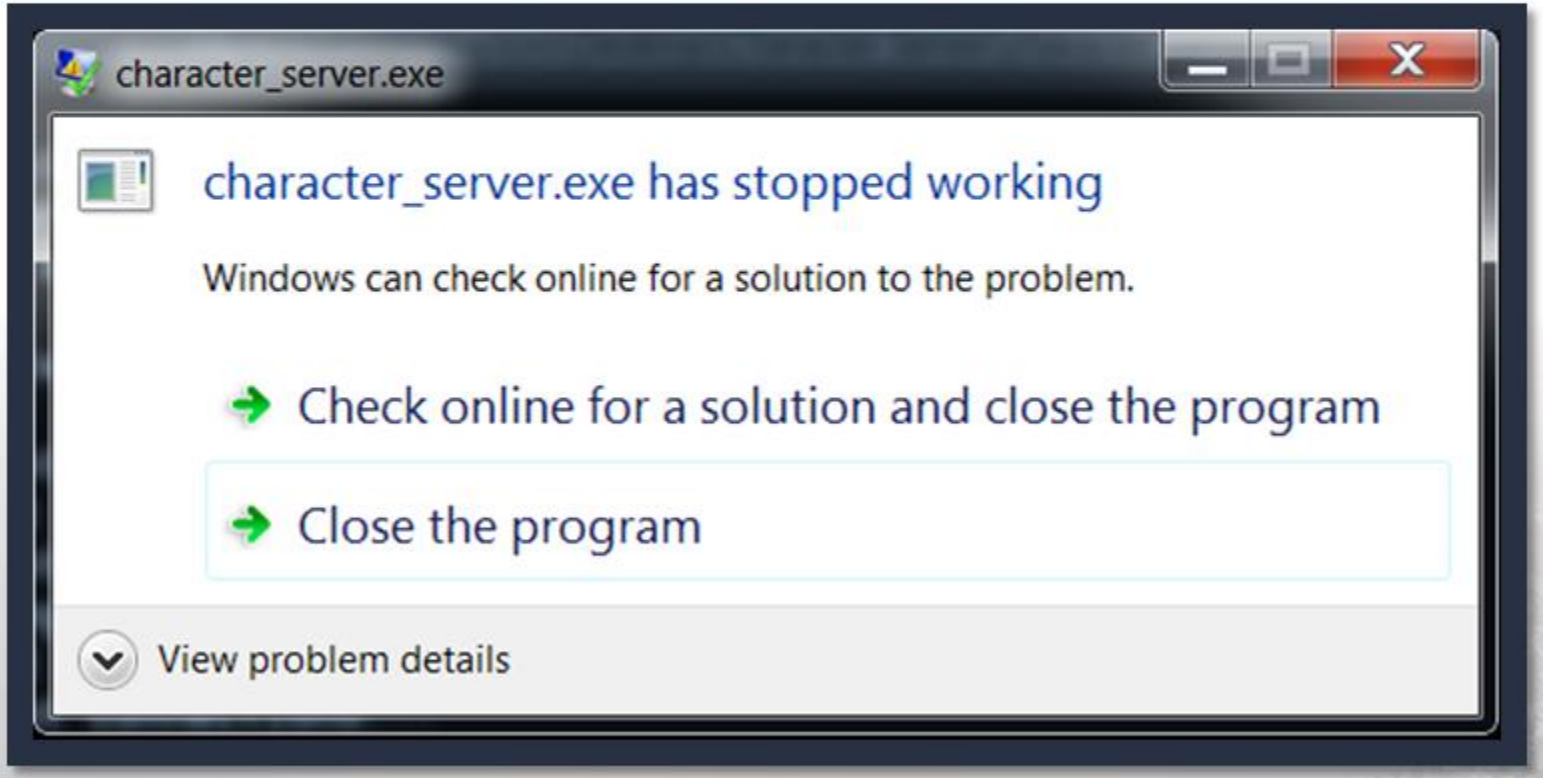
## We can definitely do it this time

- Let's increase the buffer size slightly…

```
buf = "NICK " + "A" * 100000000
```

# It fell over

- Looks like 100 million A's will work

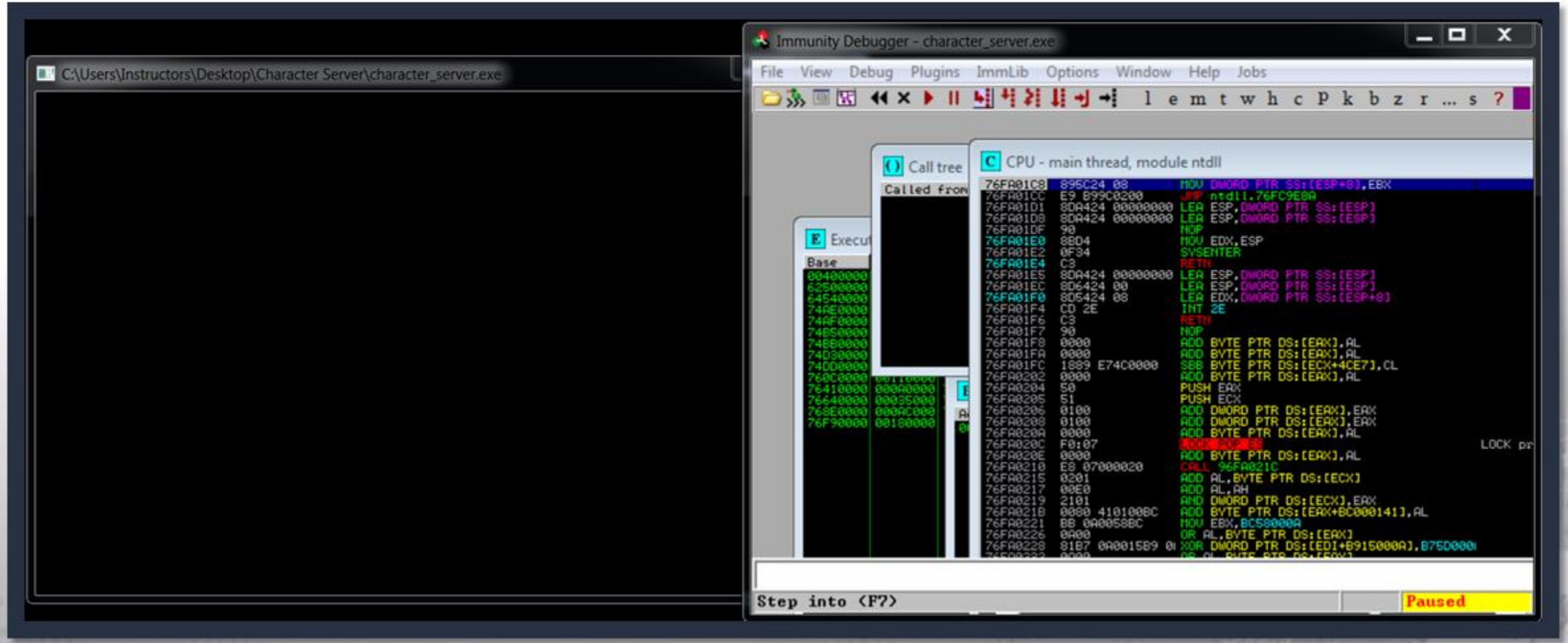


# Catching the results, a better way

- 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*

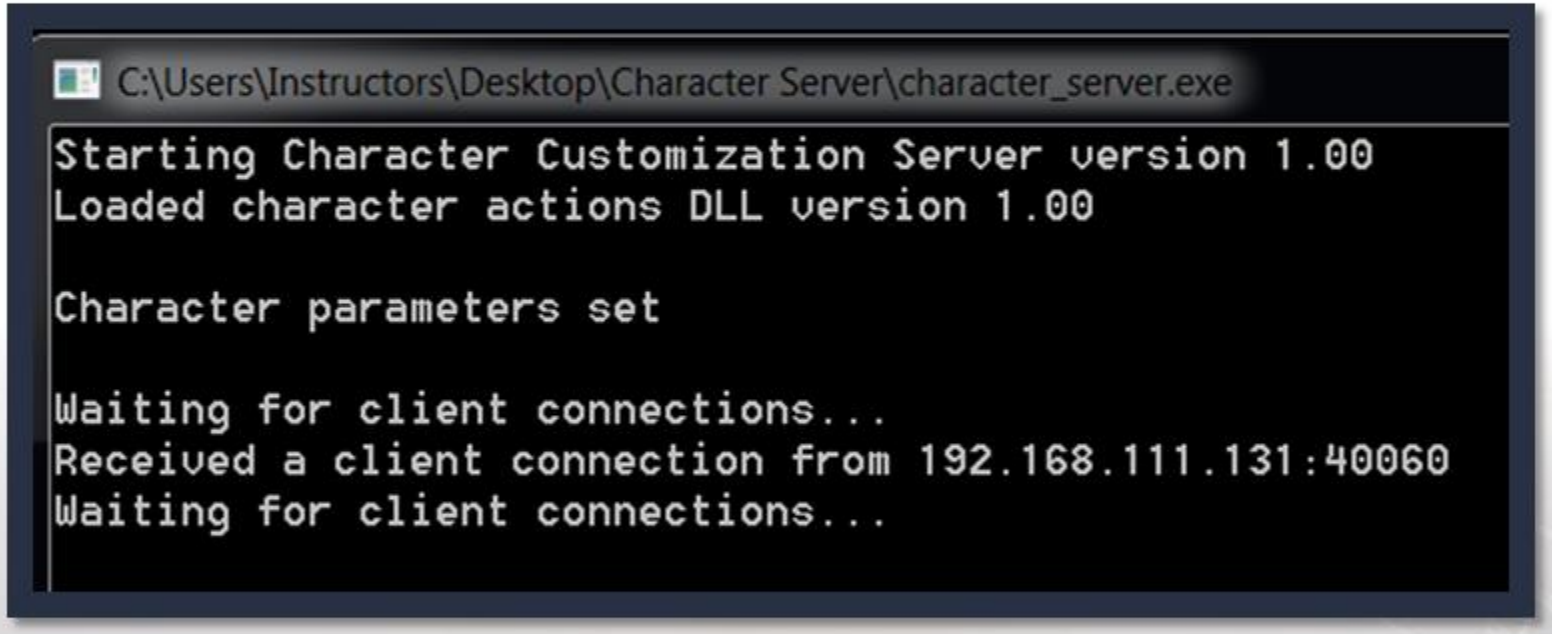
# Attaching or opening the process

- 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:

```
s_readline();
s_string("COMMAND ");
s_string_variable("BLAH");
```

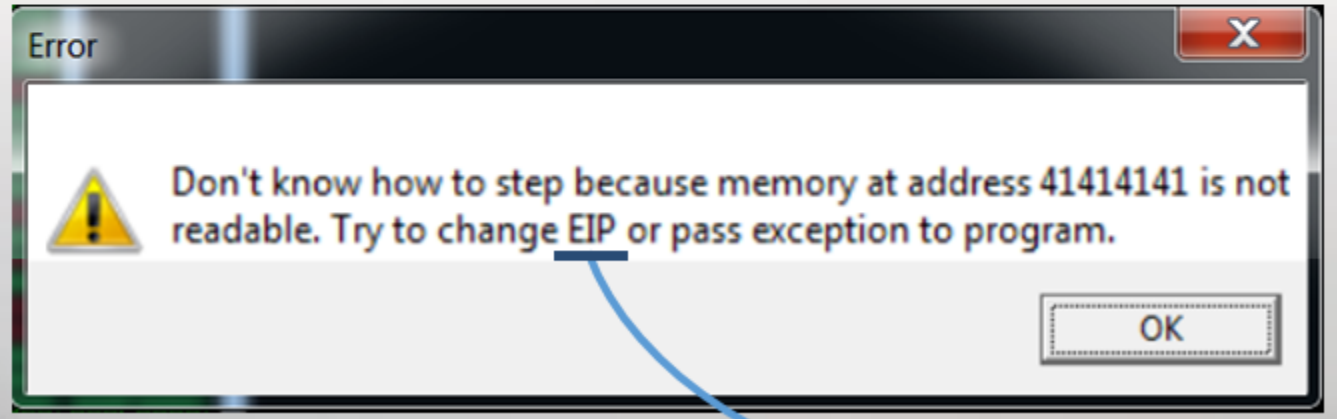
- Or, pre-configured scripts:

```
root@kali:/usr/share/spike/audits# ls
BIZTALK2000  exchange2K  IMAP                          MSSQL   PPTP        SSL
CIFS         FTPD        lotus                         ORACLE  RealServer  UPNP
COMPAQ       H323        MSContentManagementServer     POP3    SMTP
root@kali:/usr/share/spike/audits#
```

# 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 (F7) and see if EIP is corrupted




**EIP:** Extended Instruction Pointer – tells the system the address of the next command to be executed

# 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 <ip> <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?
  - What was 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 13