Exploit Development: Buffer Overflows

Course Introduction

Alexis Ahmed

Senior Penetration Tester @HackerSploit

Offensive Security Instructor @INE

Course Topic Overview

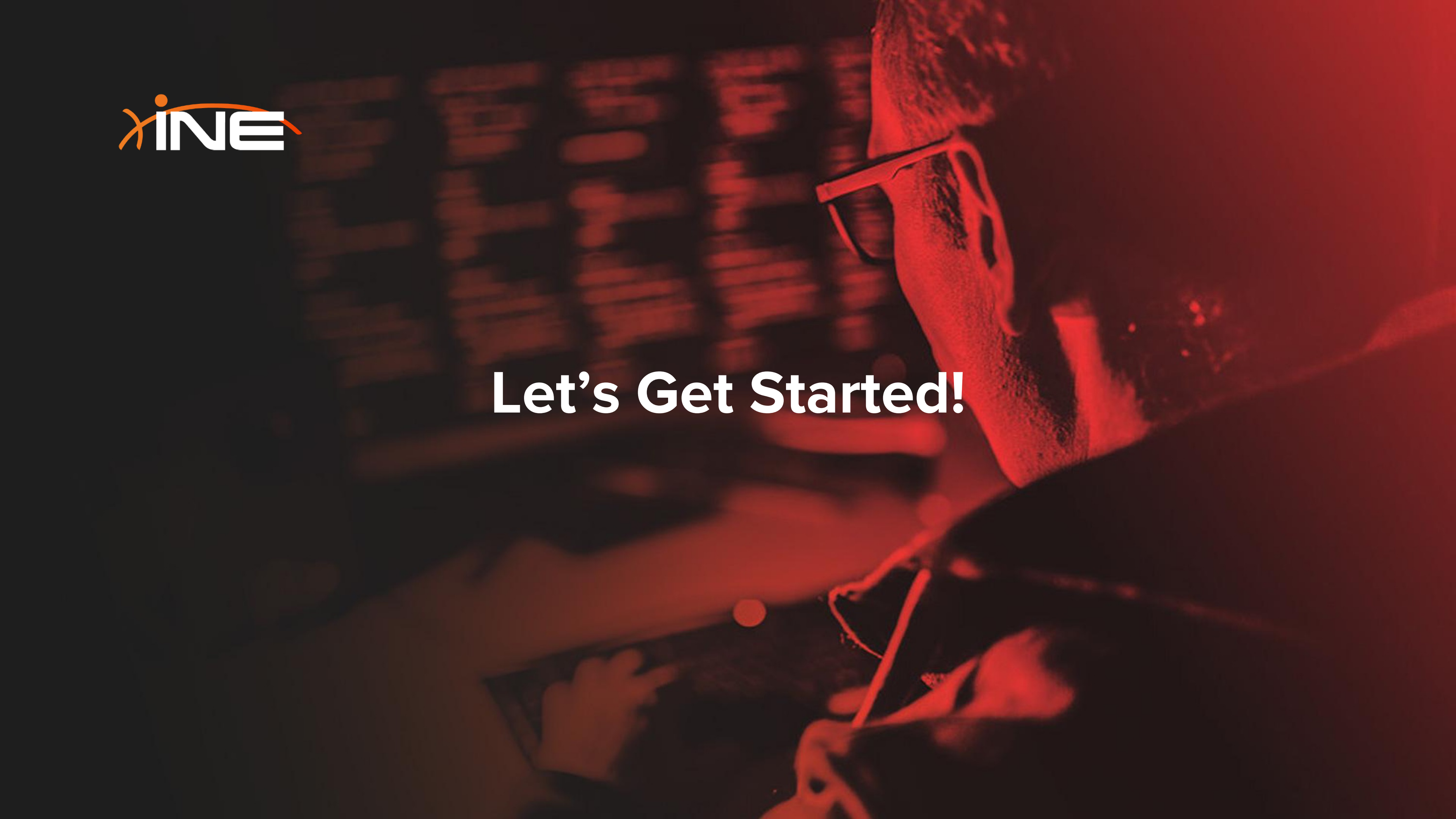
- Buffer Overflow Fundamentals
- Finding Buffer Overflow Vulnerabilities
- Stack-Based Overflows
- Windows SEH Overflow

Prerequisites

- Programming Proficiency: Students should have a strong understanding of programming concepts and experience with at least one programming language, such as C or C++, as exploit development often involves writing and analyzing low-level code.
- Operating System Fundamentals: A solid understanding of operating system concepts, particularly memory management, process execution, and system architecture, is essential for understanding how exploits work on different platforms.
- Networking Fundamentals: Basic knowledge of networking protocols (TCP/IP, UDP, HTTP, etc.) and network communication is important for understanding how exploits can be delivered over a network.
- Assembly Language: An understanding of assembly language programming is crucial for analyzing and crafting exploits, as exploits often involve manipulating machine-level instructions and memory.

Learning Objectives:

1. Understand the Basics of Memory Exploitation: Students will grasp the fundamentals of memory management in operating systems and how vulnerabilities in memory handling can lead to security exploits.
2. Identify and Exploit Buffer Overflow Vulnerabilities: Students will learn to recognize buffer overflow vulnerabilities in software applications and develop the skills to craft exploits to exploit them.
3. Analyze Stack-Based Overflow Vulnerabilities: Students will gain proficiency in analyzing stack-based overflow vulnerabilities, understanding their root causes, and exploiting them to gain control over program execution.
4. Explore Structured Exception Handling (SEH) Overflows: Students will delve into the intricacies of SEH-based overflow vulnerabilities in Windows systems, mastering techniques to manipulate SEH structures and execute arbitrary code.



Let’s Get Started!

Introduction To Buffer Overflows

Buffer Overflows

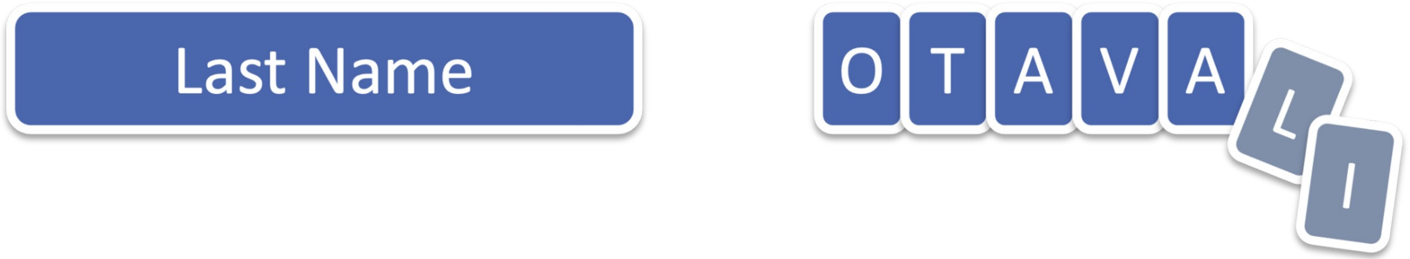
- In the previous course (System Security), we used the term “buffer overflow,” numerous times. But what exactly does that mean?
- The term buffer is loosely used to refer to any area in memory where more than one piece of data is stored.
- An overflow occurs when we try to fill more data than the buffer can handle.
- A buffer overflow can be likened to pouring 5 gallons of water into a 4-gallon bucket.

Buffer Overflows

- One common place you can see this is either online in Last Name fields of a registration form.
- In this example, the “last name” field has five boxes.

Buffer Overflows

- Suppose your last name is OTAVALI (7 characters). Refusing to truncate your name, you enter all seven characters.



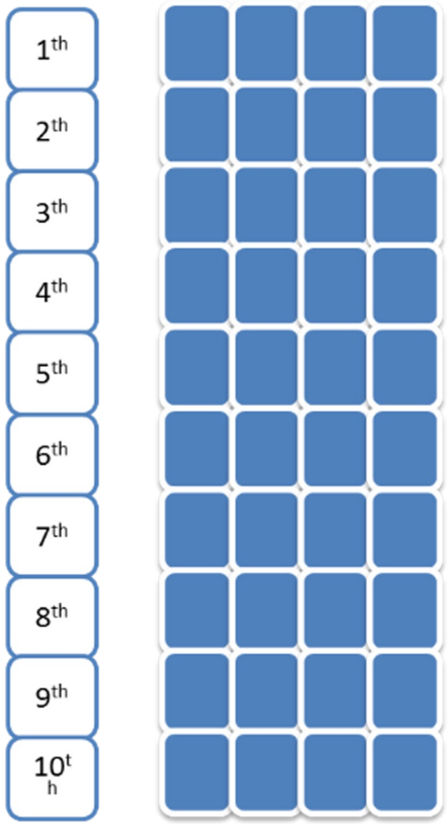
The two extra characters have to go somewhere!

Buffer Overflows

- This is what usually causes a buffer overflow, which is a condition in a program where a function attempts to copy more data into a buffer than it can hold.

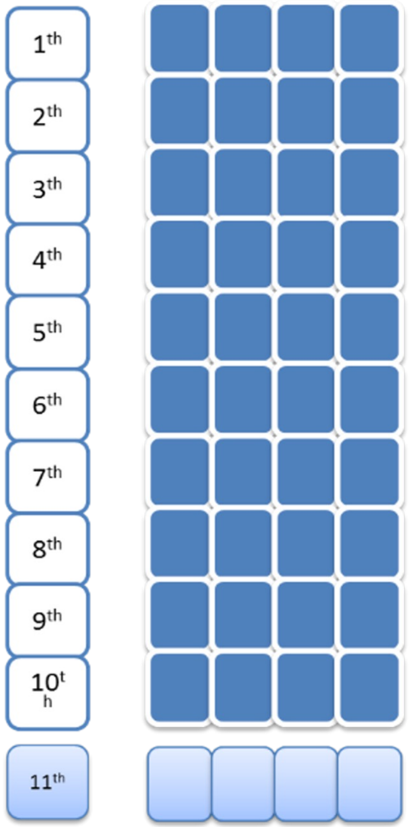
- For example, If you have allocated a specific amount of space in the stack, for instance, 10, and you exceed this by trying to copy more than 10 characters, then you will trigger a buffer overflow.

Buffer Overflows



- Suppose the computer allocates a buffer of 40 bytes of memory to store 10 integers (4 bytes per integer).
- An attacker can test for a buffer overflow by sending the computer 11 integers (a total of 44 bytes) as input.

Buffer Overflows



- Whatever was in the location after the ten 40 bytes (allocated for our buffer), gets overwritten with the 11th integer of the attacker's input.

- Remember that the stack grows backward. Therefore the data in the buffer is copied from lowest memory addresses to highest memory addresses.

Example: Vulnerable Code Sample

Vulnerable Code Sample

- Review the following code. What can you tell about it? Anything interesting?

```
int main(int argc, char** argv)
{
     argv[1] = (char*)"AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA";
     char buffer[10];
     strcpy(buffer, argv[1]);
}
```

Vulnerable Code Sample

Observations

- The array of characters (buffer) is 10 bytes long.
- The code uses the function strcpy.

Task

- Try to copy more data than the buffer can handle, using strcpy.

Demo: Triggering A Buffer Overflow

Vulnerable Code Sample

Outcome

- We can see that argv[1] contains 35 **"A"** characters, while the buffer can handle only 10. When the program runs, the exceeding data has to go somewhere, and it will overwrite something in the memory: this is a buffer overflow.

Code Outcome

- The Program Crashes.

Root Cause Analysis

- The buffer overflow vulnerability is caused by improper use of the **`strcpy`** function.

Vulnerable Code Sample

<u>**Analysis**</u>

- Without going into detail of how the function works, you should have figured out that the function <u>does not check for bounds</u>. Therefore, if the source, **`argv[1]`**, is bigger than the destination buffer/exceeds the destination buffer, an overflow occurs.

- This means that whatever was in the memory location right after the buffer, is overwritten with our input.

But what can we do with that?

Vulnerable Code Sample

- In this example, it causes the application to crash. But, an attacker may be able to craft the input in a way that the program executes specific code, allowing the attacker to gain control of the program flow.

- We will explore this in a moment.

Fixing The Vulnerable Code

Resolution

- The vulnerable code can be fixed by using a safer version of the **`strcpy`** function, called **`strncpy`** (notice the n in the function name).
- Armed with this knowledge, we can say that a safe implementation of the vulnerable code would look something like this:

```
int main(int argc, char** argv)
{
      argv[1] = (char*)"AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA";
      char buffer[10];
      strncpy(buffer, argv[1], sizeof(buffer));
      return 0;
}
```

Fixing The Vulnerable Code

- The modified code outlined in the previous slide is now safe and is not susceptible to buffer overflows. This is because the data that can be copied is limited to the limits defined by the buffer.

- In the modified code, the function will only copy 10 bytes of data from argv[1], while the rest will be discarded.

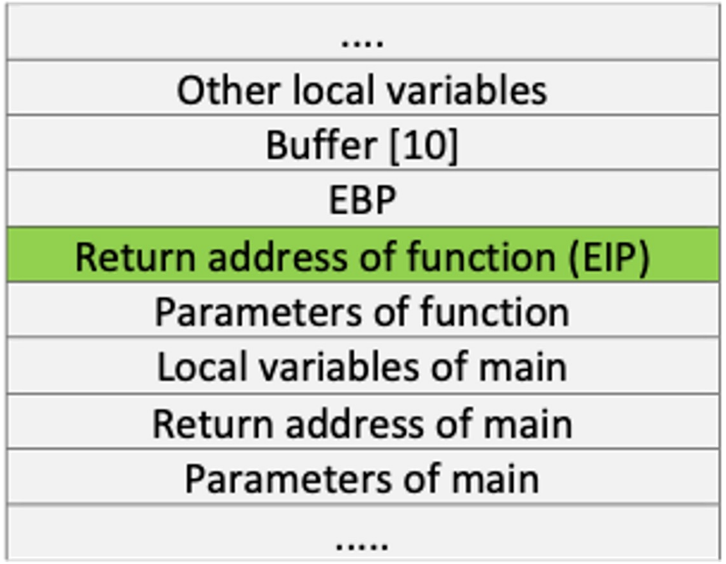
- Now that we have understood what caused the buffer overflow, we can take a closer look at what happens in the stack; this will help you understand what happens at the process memory level.

Analyzing The Stack

Analyzing The Stack

The following is the new stack frame process review:

- Push the function parameters
- Call the function
- Execute the prologue (which updates EBP and ESP to create the new stack frame)
- Allocate local variable



....
Other local variables
Buffer [10]
EBP
Return address of function (EIP)
Parameters of function
Local variables of main
Return address of main
Parameters of main
.....

Analyzing The Stack

- When the **`strcpy`** function gets executed, it starts copying our input into the memory address allocated for buffer[10].

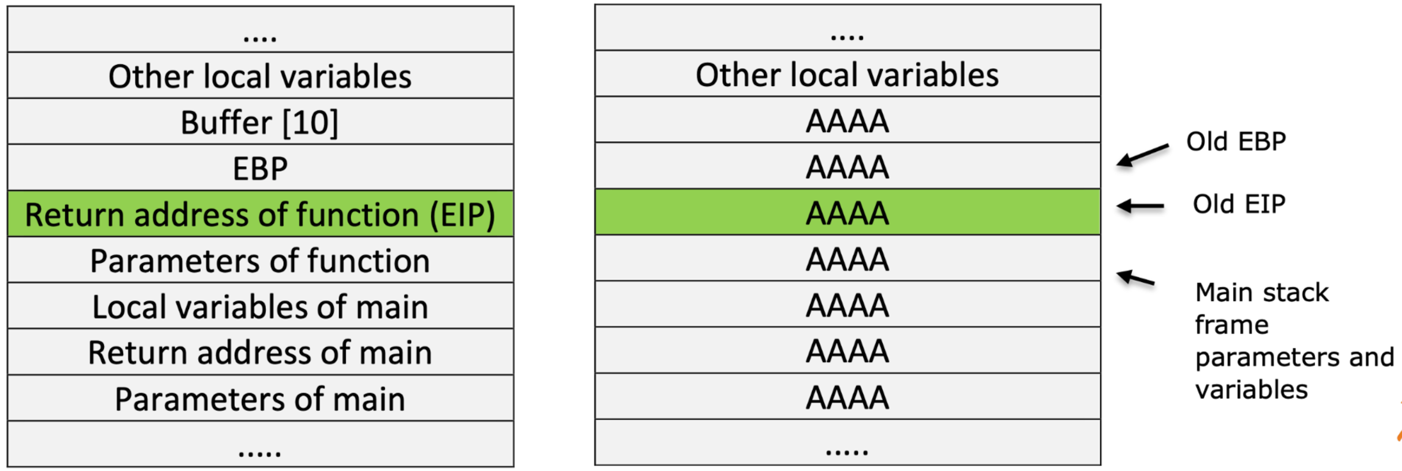
- Since there is not enough space, our input will be copied in the next memory address and will continue to fill memory addresses until there is no more input.

- While this is happening, it will also be overwriting all the data in those memory locations and causing the overflow.

Analyzing The Stack

What is getting overwritten?

- As you can see in the stack representation, this data includes the EBP, the EIP and all the other bytes related to the previous stack frame.
- Therefore, at the end of the strcpy instructions, our stack will look like the following:



Analyzing The Stack

What can a pentester do with this?

- Since the EIP has been overwritten with `AAAA`, once the epilogue takes place, the program will try to return to a completely wrong address.
- Remember that EIP points to the next instruction. An attacker can craft the payload in the input of the program to get the control of the program flow and return the function to a specific memory address location.
- This is where it is important to know memory addresses of certain registers.

Finding Buffer Overflows

Finding Buffer Overflows

- Before exploring the process of how to exploit buffer overflows and execute payloads, it is important to know how to find them.

- Any application that uses unsafe operations, such as those below (there are many others), might be vulnerable to buffer overflows.
  - strcpy
  - strcat
  - gets / fgets
  - vsprintf
  - printf
  - Memcpy

- But, it actually depends on how the function is used.

Finding Buffer Overflows

- Any function which carries out the following operations may be vulnerable to buffer overflows:
    - Does not properly validate inputs before processing/executing input
    - Does not check input boundaries
- However, buffer overflows primarily caused by unsafe languages, which allow the use of pointers or provide raw access to memory.
- All the interpreted languages such as C#, Visual Basic, .Net, JAVA, etc. are safe from such vulnerabilities.

Finding Buffer Overflows

- Moreover, buffer overflows can be triggered by any of the following buffer operations:
    - User input
    - Data loaded from a disk
    - Data from the network

- As you can imagine, if you want to manually find a buffer overflow in a large application, it will be difficult and time-consuming.

Finding Buffer Overflows

<u>BOF Hunting Techniques</u>

- When a crash occurs, be prepared to hunt for the vulnerability with a debugger (the most efficient and well-known technique).
- Some companies use cloud-fuzzing to brute-force crashing (using file-based inputs). Whenever a crash is found, it is recorded for further analysis.
- A dynamic analysis tool like a fuzzer or tracer, which tracks all executions and the data flow, can help in finding problems/errors/crashes.

Finding Buffer Overflows

- All of the techniques listed in the preceding slide can be used to discover a large number/types of vulnerabilities (such as overflows, negative indexing of an array and so on), but the challenge lies in exploiting the vulnerability.

- A large number of vulnerabilities are un-exploitable. Almost 50% of vulnerabilities are not exploitable at all, but they may lead to DOS (denial of service attacks) or cause other side-effects.

Finding Buffer Overflows With Fuzzing

Fuzzing

- Fuzzing is a software testing technique that provides invalid data, i.e., unexpected or random data as input to a program.

- Input can be in any form such as:
  - Command line
  - Parameters
  - Network data
  - File input
  - Databases
  - Shared memory regions
  - Keyboard/mouse input
  - Environment variables

Fuzzing

- In the context of finding buffer overflow vulnerabilities, fuzzing involves sending malformed or unexpected data to the target program's input (e.g., command-line arguments, file inputs, network packets) in an attempt to trigger buffer overflow errors.

- Buffer overflow vulnerabilities occur when a program writes data beyond the bounds of a buffer, potentially allowing an attacker to overwrite adjacent memory locations with malicious code or data.

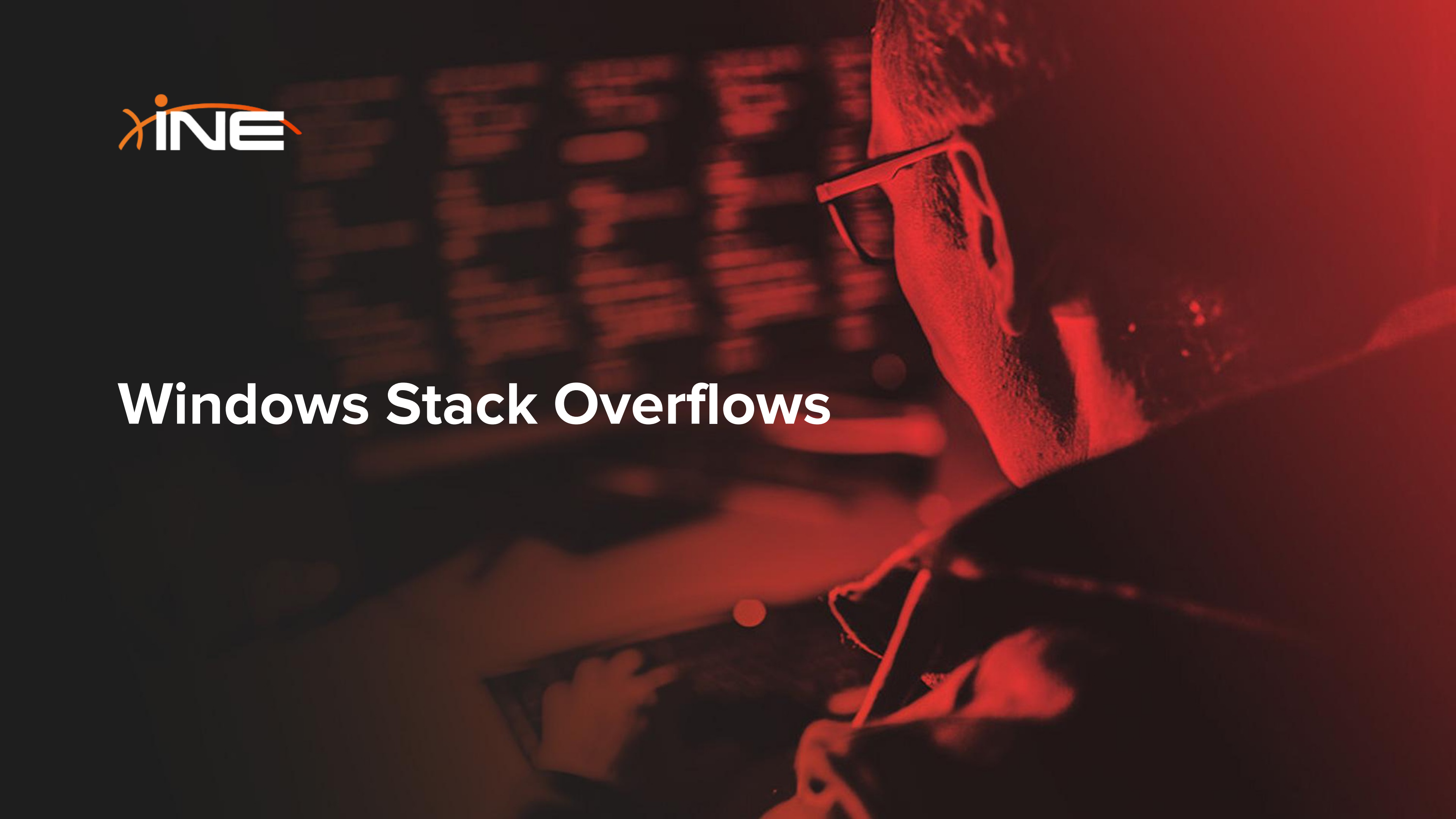
- Fuzzing aims to identify such vulnerabilities by systematically testing various input combinations to see if they cause the program to crash or exhibit unexpected behavior.

Fuzzing

- Whenever inconsistent behavior is found, all related information is collected, which will later be used by the operator to recreate the case and hunt-down/solve the problem.

- However, fuzzing is an exponential problem and is also resource-intensive, and therefore, in reality, it cannot be used to test all the cases.

Lab Demo: Finding Buffer Overflows With Fuzzing



Windows Stack Overflows

Windows Stack Overflow

- A Windows stack overflow, also known as a stack-based buffer overflow, occurs when a program writes more data to a stack-based buffer than it can hold, causing it to overwrite adjacent memory addresses on the stack.

- This vulnerability can lead to a variety of security issues, including crashes, arbitrary code execution, and privilege escalation.

How It Works

1 - Stack Memory Layout

- In Windows (as in many other operating systems), the stack is a region of memory used for storing local variables, function parameters, return addresses, and other function call-related information. The stack grows downward in memory, with the highest memory addresses at the bottom and the lowest memory addresses at the top.

2- Buffer Overflow

- A buffer overflow occurs when a program writes more data to a buffer (an array of memory) than it can hold. In the case of a stack-based buffer overflow, this buffer is typically located on the stack.
- The buffer overflow can occur due to insufficient bounds checking or improper handling of user input.

How It Works

3- Overwriting Return Address

- When a function is called in a program, the return address, which points to the instruction to be executed after the function completes, is pushed onto the stack.
- In a stack overflow attack, the attacker exploits the buffer overflow vulnerability to overwrite the return address with a malicious address pointing to code they control.

4 - Control Hijacking:

- By overwriting the return address with a malicious address, the attacker can redirect the program's control flow to execute arbitrary code of their choice.
- This code is often referred to as the "shellcode" and is typically injected into the program's memory by the attacker. Once the control flow is redirected to the shellcode, the attacker can execute commands, escalate privileges, or perform other malicious actions.

How It Works

5 -Exploitation

- With control of the program's execution flow and the ability to execute arbitrary code, the attacker can exploit the vulnerability to achieve their objectives, such as gaining unauthorized access to the system, stealing sensitive information, or launching further attacks.

  .

Structured Exception Handling (SEH)

Structured Exception Handling (SEH)

- Windows Structured Exception Handling (SEH) is a mechanism employed by the Windows operating system to manage exceptions or errors that occur during program execution.

- It provides a structured way to handle exceptions, such as access violations, divide-by-zero errors, and stack overflows.

How SEH Works

- Exception Handling Structure: SEH uses a linked list of exception handling records, known as SEH records, to manage exception handling. Each SEH record contains information about a specific exception handler, including a pointer to the handler function.

- Registration: When an application wishes to handle exceptions using SEH, it registers one or more exception handlers by adding SEH records to the SEH chain. These records are typically added using the __try and __except keywords in C/C++ code.

How SEH Works

- Exception Handling: When an exception occurs during program execution, the Windows kernel walks through the SEH chain, searching for an appropriate exception handler to handle the exception. If a matching handler is found, the associated handler function is executed, allowing the program to respond to the exception in a controlled manner.

Windows SEH Overflows

- Windows SEH overflows, also known as SEH-based buffer overflows, occur when an attacker exploits a vulnerability in an application's exception handling mechanism to gain unauthorized access or execute arbitrary code.

Windows SEH Overflows

- Vulnerability Identification: The attacker identifies a vulnerability in the application that allows them to overwrite an exception handler's SEH record. This vulnerability may be due to insufficient bounds checking or improper exception handling.

- SEH Record Overwrite: The attacker exploits the vulnerability to overwrite the SEH record of an exception handler with a malicious address pointing to their shellcode or payload.

- Exception Triggering: The attacker triggers an exception condition within the vulnerable application, such as by providing specially crafted input or triggering a specific code path.

Windows SEH Overflows

- Exception Handling: When the exception occurs, the Windows kernel walks through the SEH chain to locate the exception handler. Instead of finding a legitimate handler, it encounters the overwritten SEH record pointing to the attacker's payload.

- Code Execution: The attacker's payload is executed, allowing them to gain control of the application's execution flow. From there, they can execute arbitrary code, escalate privileges, or achieve other malicious objectives.

Windows SEH Overflow (EasyChat)

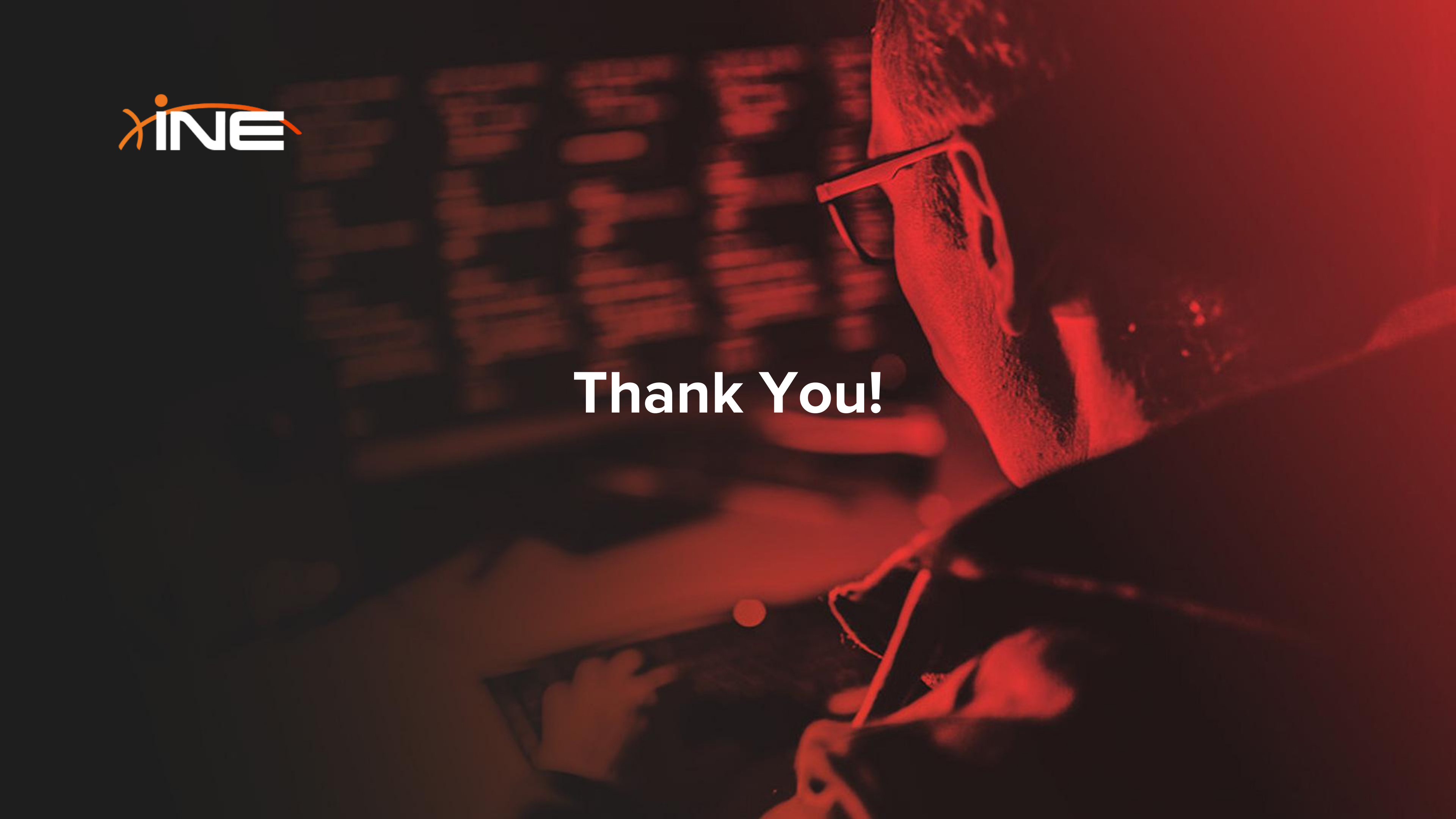
Lab Demo: Windows SEH Overflow (EasyChat)

Exploit Development

Course Conclusion

Learning Objectives:

1. Understand the Basics of Memory Exploitation: Students will grasp the fundamentals of memory management in operating systems and how vulnerabilities in memory handling can lead to security exploits.
2. Identify and Exploit Buffer Overflow Vulnerabilities: Students will learn to recognize buffer overflow vulnerabilities in software applications and develop the skills to craft exploits to exploit them.
3. Analyze Stack-Based Overflow Vulnerabilities: Students will gain proficiency in analyzing stack-based overflow vulnerabilities, understanding their root causes, and exploiting them to gain control over program execution.
4. Explore Structured Exception Handling (SEH) Overflows: Students will delve into the intricacies of SEH-based overflow vulnerabilities in Windows systems, mastering techniques to manipulate SEH structures and execute arbitrary code.



Thank You!

EXPERTS AT MAKING YOU AN EXPERT

INE