CSE 2010 SECURE CODING
ASSIGNMENT-2
19BCN7186 /02.05.2021 /NISHANTH R
SLOT- H+TH

1]What is DEP? How to bypass DEP, explain with a suitable example.

DEP works by routinely scanning the memory heaps and stacks for actions of loading data into the memory. The hardware enforced DEP mechanism uses theCPU to mark all memory locations that are flagged with an attribute value for non-execution. Once an abnormality is detected in these locations in terms of code execution, an exception is sent to the primary OS security mechanism. Software enforced DEP only checks for an exception within the functions table of the primary application. This provides protection against security esploits like buffer overflow.

Bypassing DEP:

1.) Turn on DEP

Though DEP is already enabled by default, but just to be sure let's check that it's on.

Navigate to: Control Panel -> System and Security -> System -> Advanced System Settings

Then choose "Turn on DEP for all programs and services except those I select" if not already

Choose Apply and Okay everywhere and restart the system.

2.) Setting up the exploit development environment

People with experience in stack based Buffer Overflow exploit development will be familiar of these interim steps.

a.) Start the testing Windows machine, wherein we will debug the vulnerable application to twerk and develop our fully functional exploit.

b.) Make sure the vulnerable application is installed and running properly.

c.) Ensure Immunity debugger is working properly and mona.py is present in the PyCommands folder of Immunity Debugger Application.

3.) Finding the Offset

Now that everything is up and running let's move on to the fun part — the exploit development process.

The application which we are using is called vulnserver, which as the name suggest is vulnerable.

In vulnserver TRUN command has been found to be vulnerable to stack based buffer overflow, which in layman's terms means that the application will crash when an input string of long length that the application can't handle is sent through the TRUN command. To be a bit more technical, since the application has no boundation on the length of input that it can receive, so the memory space (buffer) and the EIP (instruction pointer) gets overwritten.

To verify this, let's send a string of say 3000 from the attacker machine to the application to ensure that it is vulnerable:

```
#!/usr/bin/python

import socket,sys

host="192.168.2.135"

port=9999

buffer = "TRUN /.:/" + "A" * 3000

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

expl.connect((host, port))

expl.send(buffer)

expl.close()
```

Programming to crash the application.

We aim to get control of the EIP and point it to a location where our shellcode resides.

For that we aim to the find the length of string (offset) after which the EIP is overwritten.

a.) We are going to utilise metasploit's scripts to figure this out.

Run the following command in kali terminal to generate a random string of length 3000

```
/usr/share/metasploit-framework/tools/pattern_create.rb -l 3000
```

Now, instead of the AAAs that we were sending to crash the application, we are going to send this random string of same length and find out the character being written in EIP.

We restart the application from Immunity Debugger (Debugger->Restart) and run the below script from the attacker machine.

EXAMPLE:

```
#!/usr/bin/python

import socket

server = 'xx.xx.xx.xx'

sport = 9999

prefix =
'Aa0Aa1Aa..... Du2Du3Du4Du5Du6Du7Du8Du9Dv0Dv1Dv2Dv3Dv4Dv5Dv6Dv7Dv8Dv9'

attack = prefix

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

connect = s.connect((server, sport))

print s.recv(1024)

print "Sending attack to TRUN . with length ", len(attack)

s.send(('TRUN .' + attack + '\r\n'))

print s.recv(1024)

s.send('EXIT\r\n')

print s.recv(1024)

s.close()
```

The application will crash again but this time EIP will be overwritten with a part of the random string that we sent from the attacker machine.

b.) Again we will use metasploit to figure out the exact offset.

Run the following command from the Kali terminal:

```
/usr/share/metasploit-framework/tools/pattern_offset.rb -q 386f4338 -l 3000
```

Replace the text highlighted in yellow with whatever characters EIP was overwritten with.

The output will tell us the offset enabling us to write whatever we wish to in the EIP.

For vulnserver it came out to be 2006. This means that after 2006 characters the next four characters overwrite the EIP.

Now the payload which we will send on to the victim will be similar to that of buffer overflow.

```
padding = 'F' * (3000–2006–4 — len(padding))

prefix = A*2006

attack = prefix + '\x42\x42\x42\x42'+padding
```

4.) Developing ROP Chain

Now that we have control of EIP we point it to the address of whatever instruction that we want to execute next. For a normal Buffer Overflow the EIP would have pointed to a JUMP instruction that will further jump to our shellcode present in the stack giving us a shell back from the victim system.

But with DEP turned on, whenever the exploit tries to execute some instruction in the stack an access violation occurs, so the normal Buffer Overflow exploit is useless for now.

To bypass this we are going to build the ROP chain.

Though the whole ROP concept is sounds overwhelming at first, the actual execution process is not difficult.

We just need to run the following command from the Immunity Debugger instruction bar.

!mona rop -m *.dll -cp nonul

Then wait for the process to end, which will take roughly around 3 minutes. Mona will meanwhile go through all the dlls (*.dll) and build a chain of usable gadgets.

We are going to use the python code for VirtualProtect() from rop_chains.txt and exploit.

But before moving on with our ready to use code let's pause and try to understand what exactly is happening.

VirtualProtect() will turn off DEP for a part of memory, so the code placed in that part of the memory can execute

VirtualProtect() requires five arguments:

IpAddress: Points to a region for which DEP has to be turned off, this will be the base address of the shell code on stack.

dwsize: Size of the region for which DEP has to be turned off

flNewProtect: Memory protection constant to which the protection level has to be changed to

IpflOldProtect: points to a variable that will receive the previous access protection value

ReturnAddress: pointer to the location where VirtualProtect() will return after executing i.e. our shellcode

Now ROP gadgets will be used to develop the above mentioned arguments that VirtualProtect() needs, set the values as required and execute the function.

Let's have a look at the ROP function generated by mona and try to understand how it works.

Lines 11,12,13,14 — dwSize of 0X201 was put in EAX and then transferred to EBX

Lines 15,16,17,18 — The Memory Protection constant 0x40 (read-write privileges) was put in EAX then transferred to EDX for flNewProtect

Lines 19,20 — A pointer to a writable location has been set in ECX for IpflOldProtect

Lines 7,8 and 21,22 — ESI and EDI were populated for PUSHAD call to execute.

Lines 9,10 — EBP was set to a jump instruction, for ReturnAddress.

Lines 5 and 6 — ECX was set to call VirtualProtect()

Lines 23,24,25 — A PUSHAD call is placed in EAX at the end, which will flush all the values that were put in the register onto the stack.

Now that our code is ready, let's try to execute some malicious code on the victim machine.

Let's try opening up a calculator.

We place the malicious code along with the ROP chain in our exploit.

As seen in the code below, we first set calc to a malicious code that will open up a calculator.

Followed by the declaration of the ROP function generated by mona,Then we call the create_rop_chain function, remove bad characters (\x00 for vulnserver) and store it in the variable rop_chain.

Now that we have declared all the important stuff we just need to piece together our payload and send it to the victim machine. Which we are doing in the following lines:

padding = 'F' * (3000–2006–16 — len(shellcode))

attack = prefix + rop_chain +nops + calc + padding

Our prefix is A*2006 so the EIP will be pointing to the ROP chain code. The ROP chain code will execute the VirtualProtect() API, which in turn will allocate a memory location with DEP turned off, where we will place our malicious code.

The we append our malicious code with nops and add padding at end to ensure that payload length is 3000.

Then we send out the exploit and as evident from the image below the calculator will open up in the victim windows machine.

So, our exploit was successfully able to bypass DEP and execute commands on the victim machine.