Writing Metasploit Plugins

from vulnerability to exploit

Saumil Shah

ceo, net-square

Hack In The Box 2006, Kuala Lumpur

# who am i

```
# who am i
16:08  up  4:26, 1 user, load averages: 0.28 0.40 0.33
USER     TTY      FROM                LOGIN@  IDLE WHAT
saumil   console  -                  11:43    0:05 bash
```

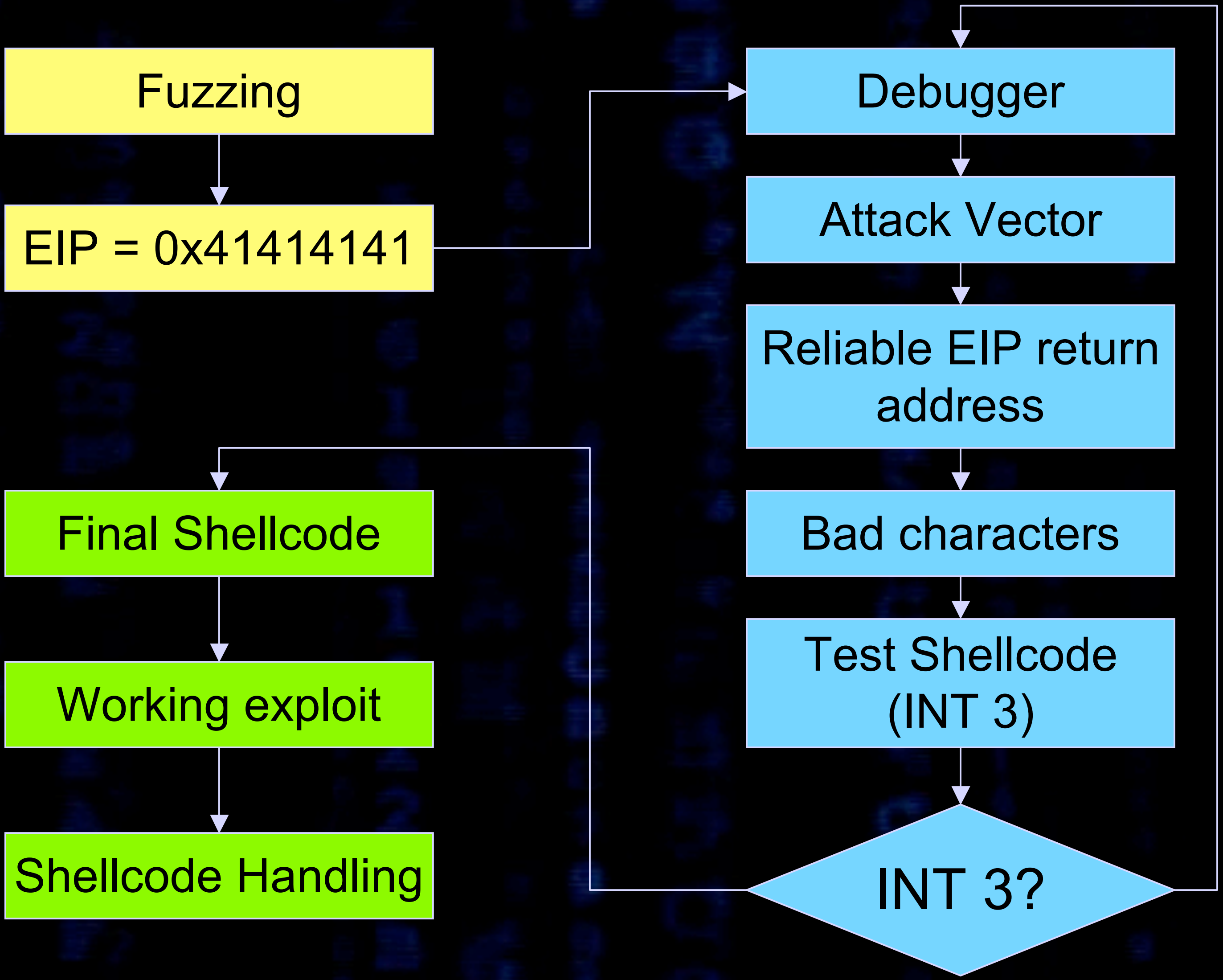
- Saumil Shah - “krafty”

  ceo, net-square solutions

  saumil@saumil.net

  author: “Web Hacking - Attacks and Defense”

From Vulnerability to Exploit



Fuzzing

EIP = 0x41414141

Debugger

Attack Vector

Reliable EIP return address

Bad characters

Test Shellcode (INT 3)

INT 3?

Final Shellcode

Working exploit

Shellcode Handling

© Saumil Shah

Stack Overflows

- Error condition when a larger chunk of data is attempted to be written into a smaller container (local var on the stack).

```
char buffer[128];
strcpy(buffer, argv[1]);
```

- What will happen if “argv[1]” is more than 128 bytes?

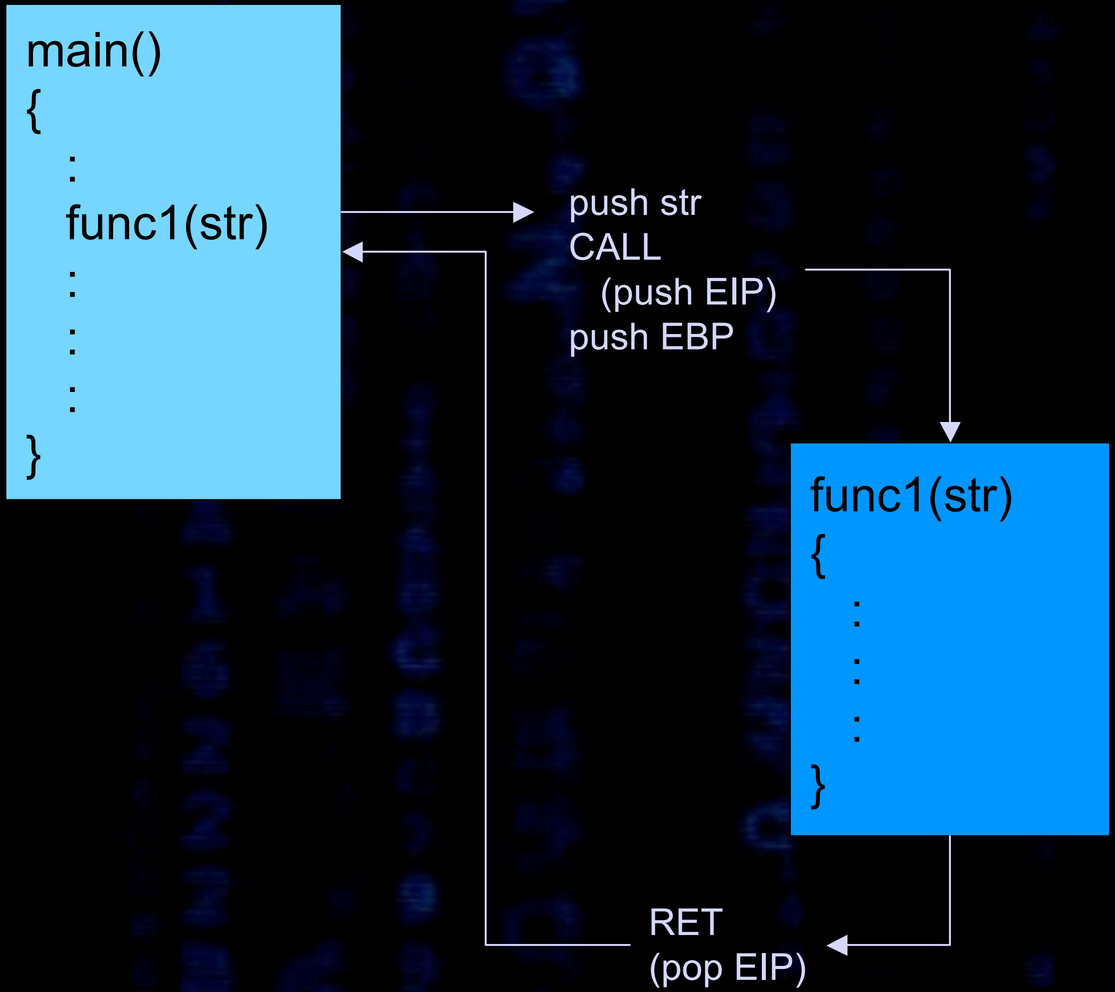
Post mortem debugging

- Register dump after a stack overflow:

```
(gdb) info registers
esp            0xbffffb24       -1073743068
ebp            0x41414141       1094795585
esi            0x4000ae60       1073786464
edi            0xbffffb74       -1073742988
eip            0x41414141       1094795585
```

- EIP's value is "0x41414141", i.e. "AAAA"
- EIP got overwritten with bytes from the overflowed buffer.

Calling a function



```
main()
{
   :
   func1(str)
   :
   :
   :
}
```

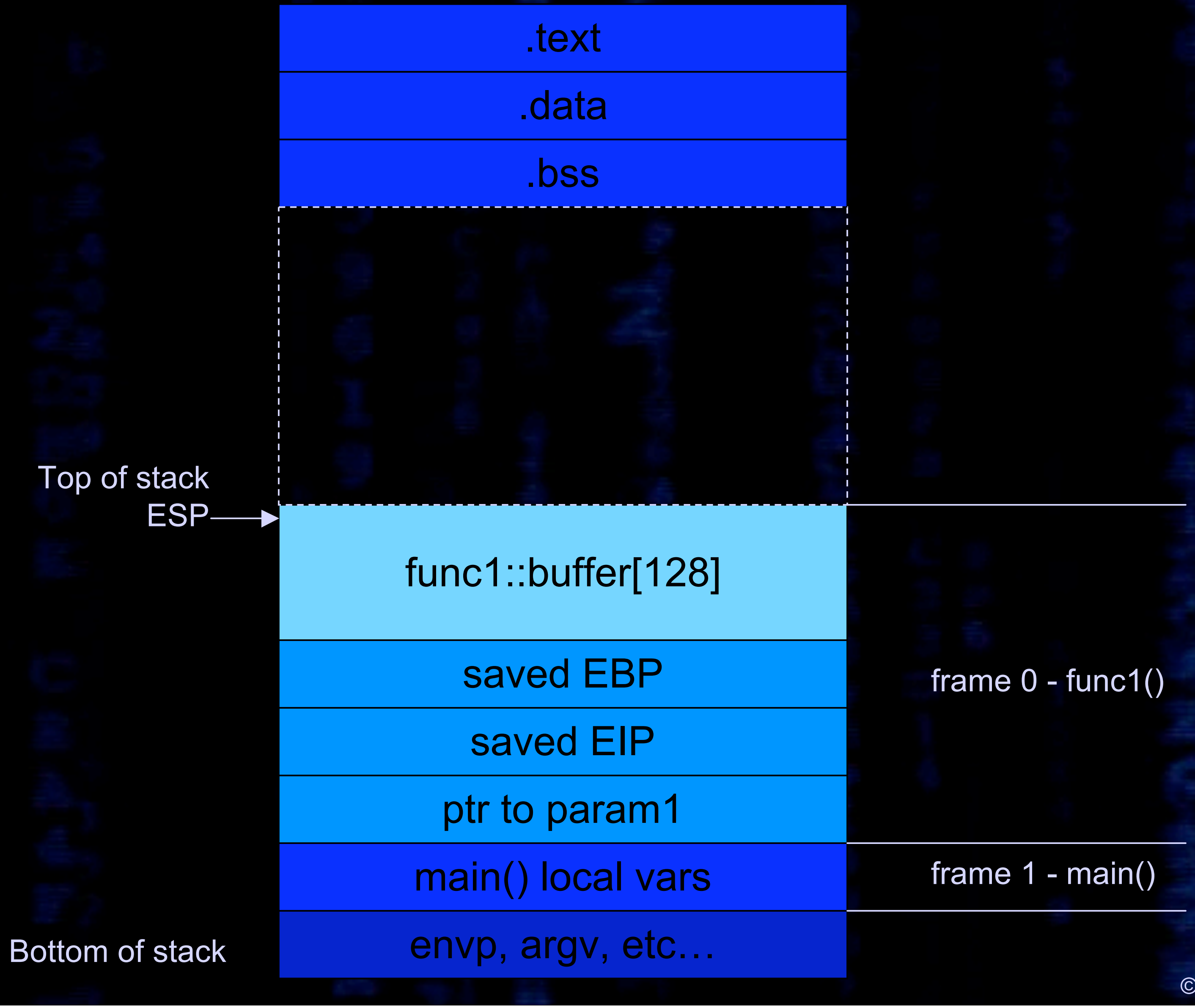
push str
CALL
    (push EIP)
push EBP

```
func1(str)
{
   :
   :
   :
}
```

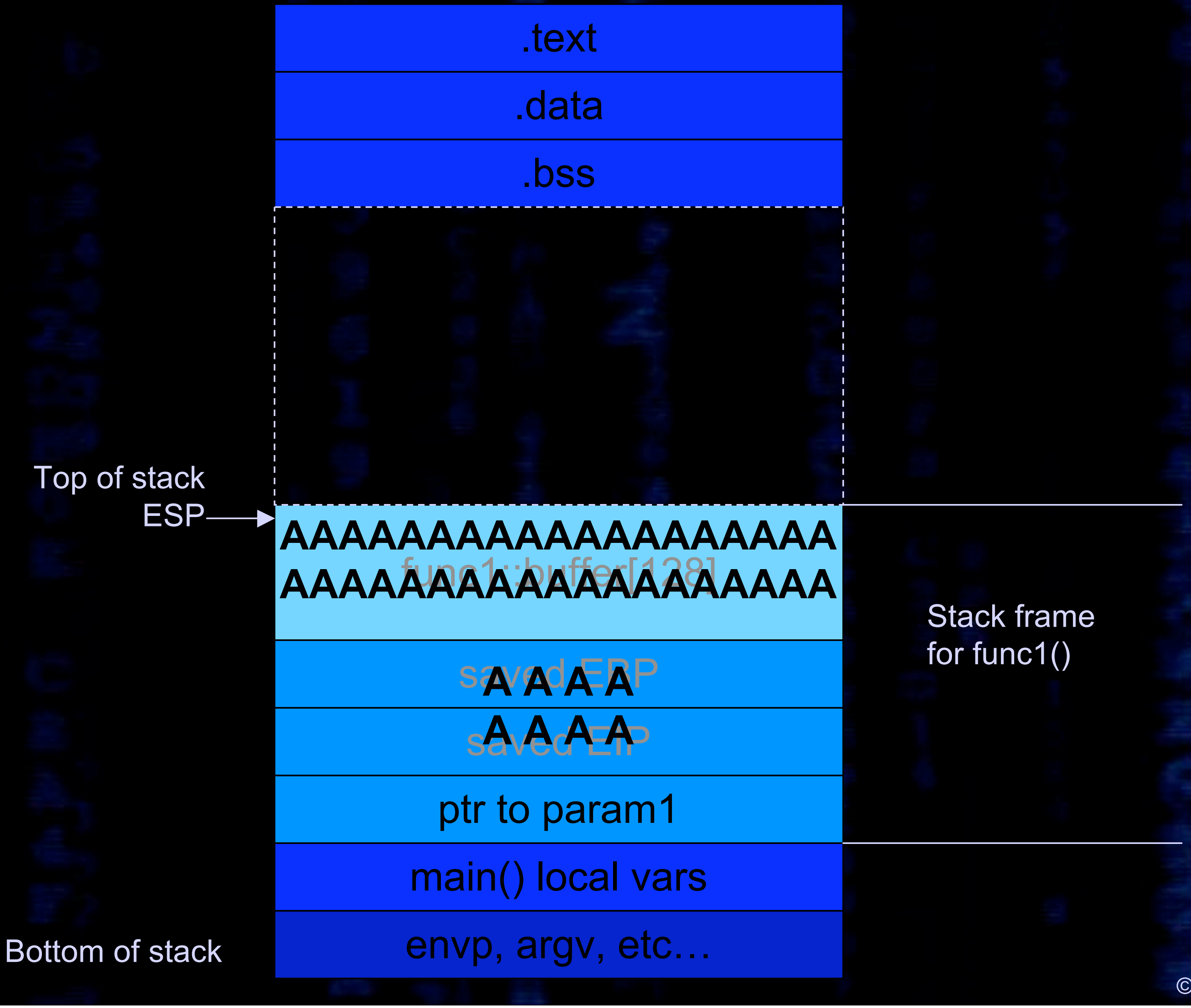
RET
(pop EIP)

© Saumil Shah

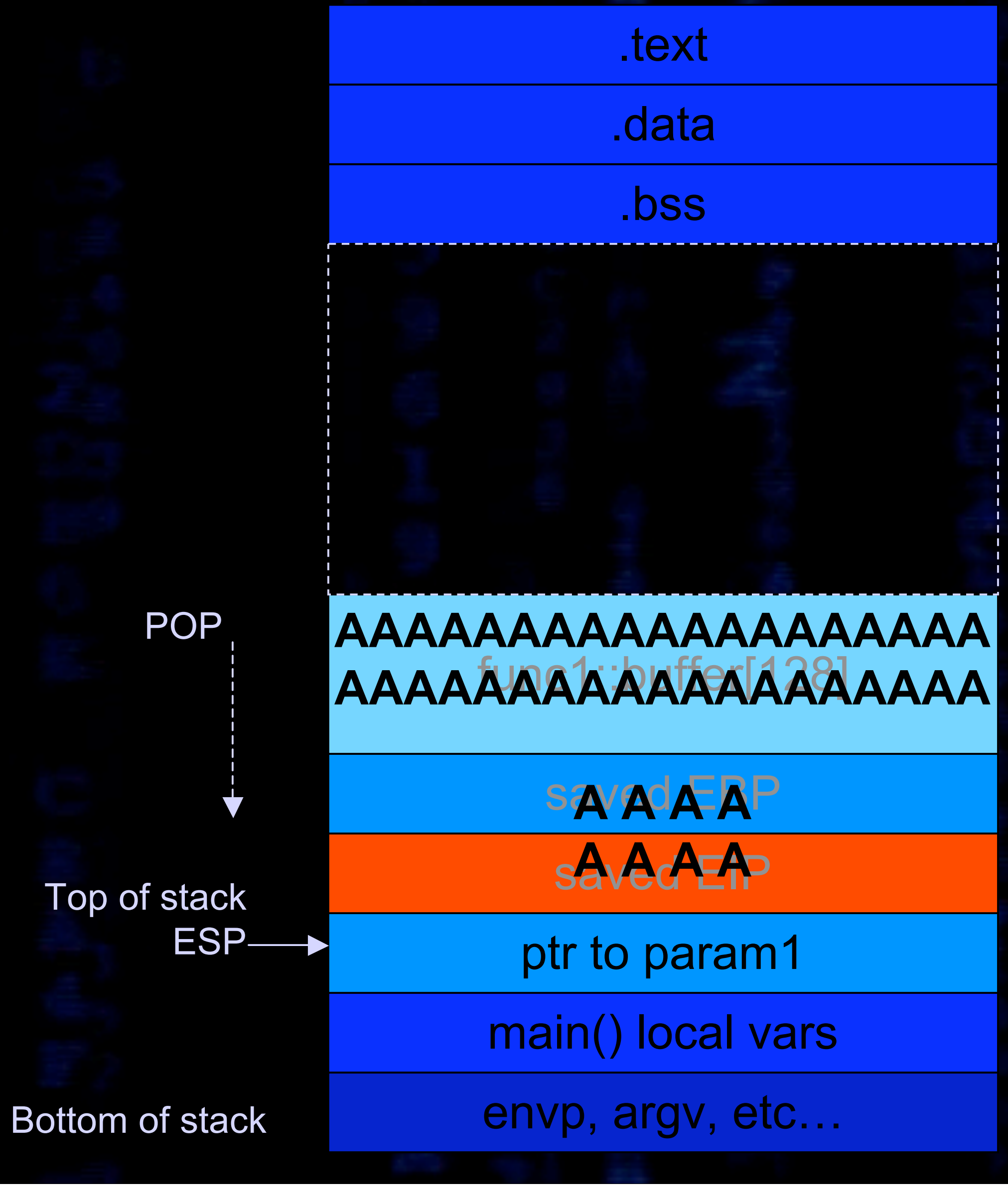
victim's Memory Map - before



victim’s Memory Map - after



The Stack Overflowed



.text

.data

.bss

AAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAA

func1::buffer[128]

saved EBP

A A A A

saved EIP

A A A A

ptr to param1

main() local vars

envp, argv, etc…

POP

Top of stack
ESP

Bottom of stack

when func1 returns
EIP will be popped
EIP = 0x41414141
(“AAAA”)

Registers after the Stack Overflow

- After func1() returns, EIP and EBP are popped off the stack

```
(gdb) info registers
esp            0xbffffa24       -1073743324
ebp            0x41414141       1094795585
esi            0x4000ae60       1073786464
edi            0xbffffa74       -1073743244
eip            0x41414141       1094795585
```

- We have control of the instruction pointer.

Controlling EIP

- Vulnerabilities may lead to EIP control.
- “Where do we want to go…. today?”
- Can we inject our own code, and make EIP jump to it?
- And, where do we inject our code?

Introducing Metasploit

- An advanced open-source exploit research and development framework.
- http://metasploit.com
- Current stable version: 2.6
  - Written in Perl, runs on Unix and Win32 (cygwin)
  - 160+ exploits, 77 payloads, 13 encoders
- Brand new 3.0 beta2
  - Complete rewrite in Ruby

Introducing Metasploit

- Generate shellcode.
- Shellcode encoding.
- Shellcode handlers.
- Scanning binaries for specific instructions:
  - e.g. POP/POP/RET, JMP ESI, etc.
- Ability to add custom exploits, shellcode, encoders.
- …and lots more.

EIP = 0x41414141

- How do we determine which 4 bytes go into EIP?
- Use a cyclic pattern as input:

```
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1
Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3
Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5
Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5............
```

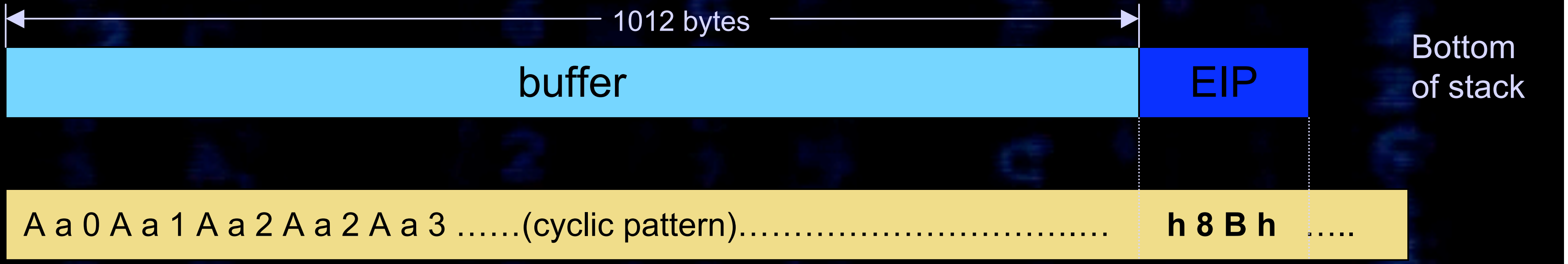
- Metasploit’s Pex::Text::PatternOffset()
- Generate patterns, find substring.

Distance to EIP

- Use Metasploit's patternOffset.pl

```
krafty:~/metasploit$ perl sdk/patternOffset.pl 0x68423768 2000
1012
```

- Based on what EIP gets overwritten with, we can find the "distance to EIP" with this pattern.


1012 bytes
buffer
EIP
Bottom of stack
A a 0 A a 1 A a 2 A a 2 A a 3 ......(cyclic pattern)..................................
h 8 B h
.....

© Saumil Shah

Getting Control of Program Counter

- Stack Overflows
  - Direct Program Counter overwrite
  - Exception Handler overwrite
- Format String bugs
- Heap Overflows
- Integer Overflows

- Overwrite pc vs. “what” and “where”

Enter Shellcode

- Code assembled in the CPU’s native instruction set.
- Injected as a part of the buffer that is overflowed.
- Most typical function of the injected code is to “spawn a shell” - ergo “shellcode”.
- A buffer containing shellcode is termed as “payload”.

Writing Shellcode

- Need to know the CPU’s native instruction set:
  - e.g. x86 (ia32), x86-64 (ia64), ppc, sparc, etc.
- Tight assembly language.
- OS specific system calls.
- Shellcode libraries and generators.
- Metasploit Framework.

Injecting the shellcode

- Easiest way is to pack it in the buffer overflow data itself.
- Place it somewhere in the payload data.
- Need to figure out where it will reside in the memory of the target process.

Where do you want to go…today?

- EIP can be made to:
  - Return to Stack

    Jump directly into the payload.

    (reliability issues - addr jitter, stack protection)

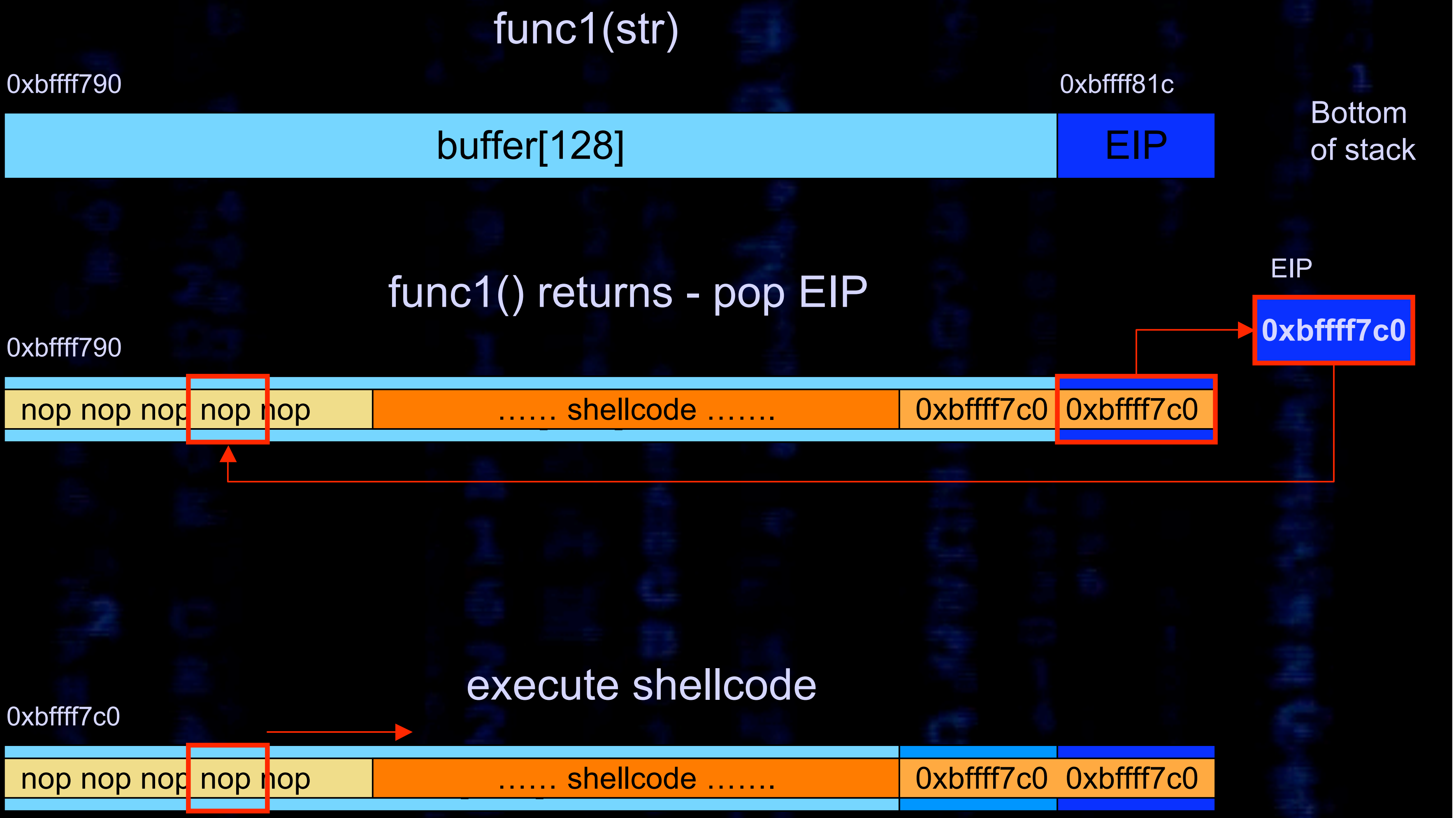
  - Return to Shared library

    Jump through registers.

    Requires certain conditions to be meet.

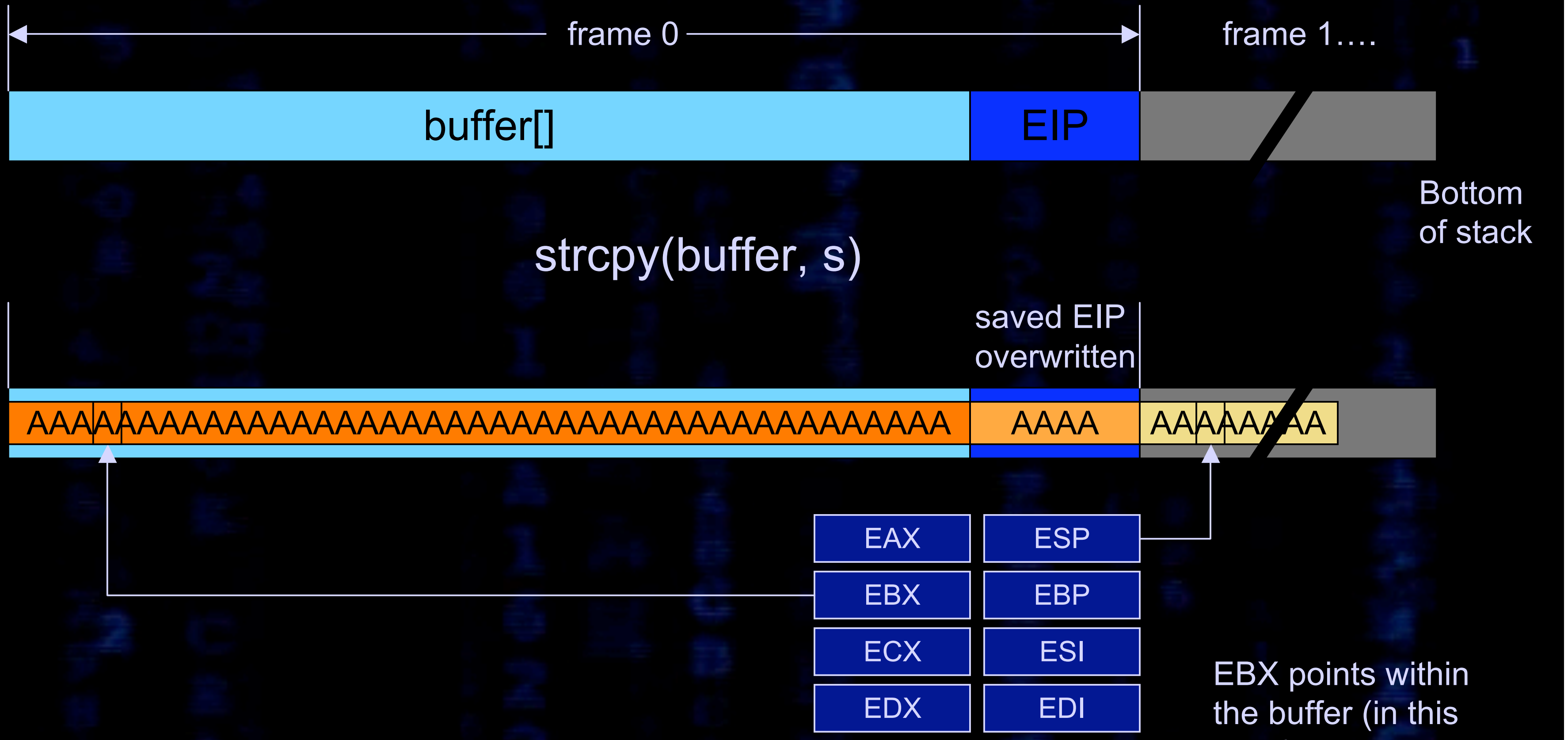
    (highly stable technique)

Return to Stack



© Saumil Shah

Jump through Register

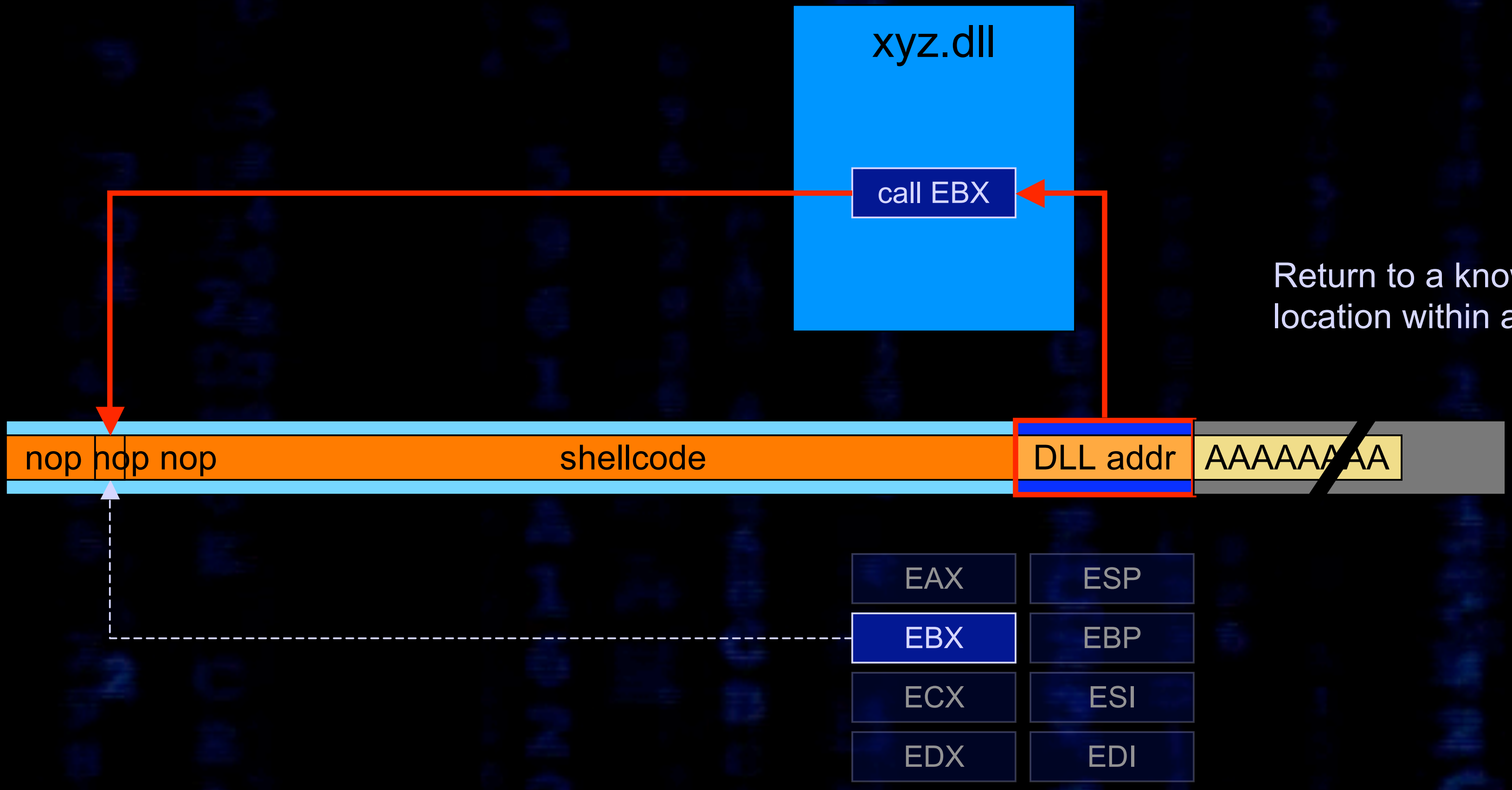


EBX points within the buffer (in this case)

ESP points beyond the saved EIP

© Saumil Shah

Jump through Register

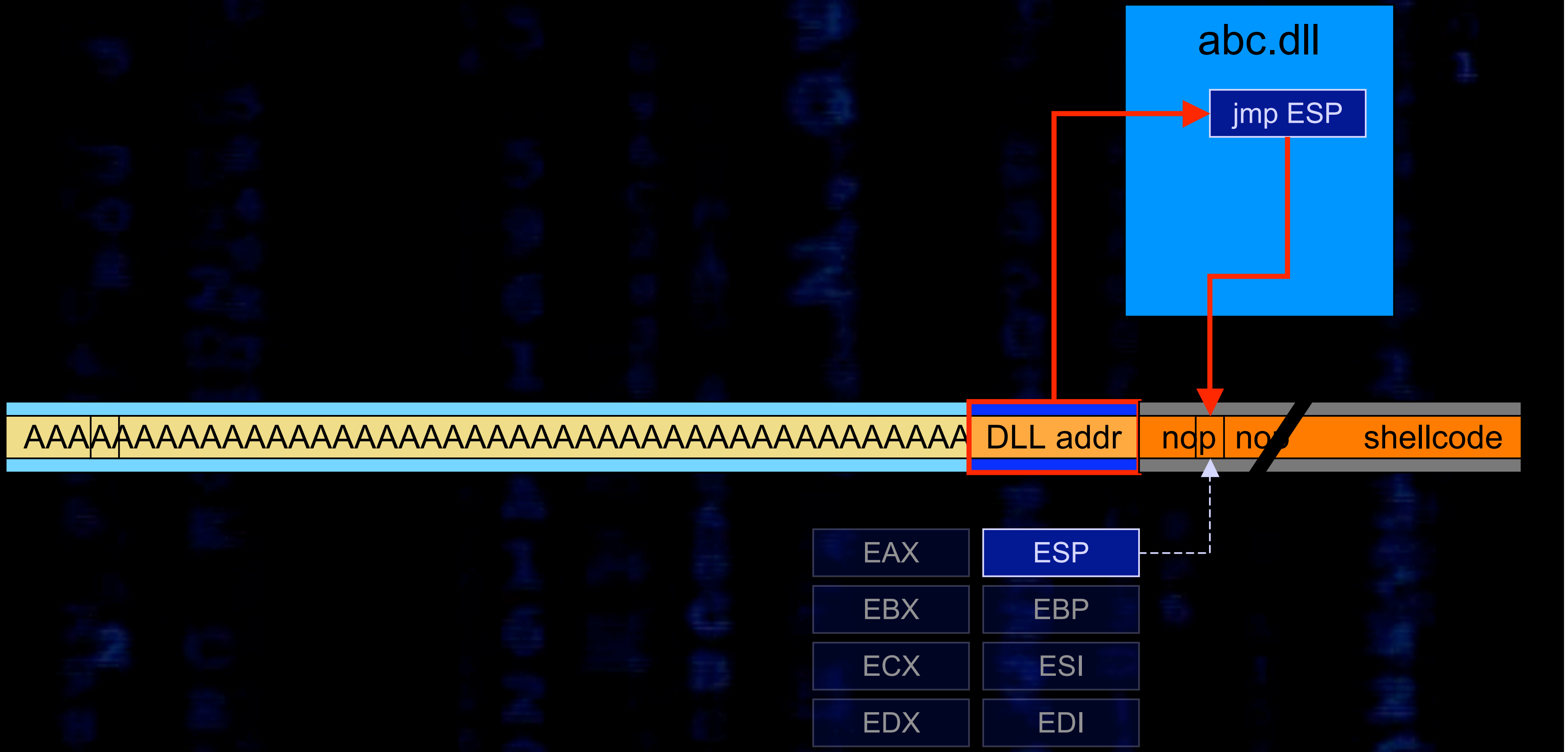


Return to a known location within a DLL

shellcode at the beginning of the buffer

© Saumil Shah

Jump through Register



shellcode at the end of the buffer

Looking for CALL or JMP instructions

- We need to find locations in memory which contain CALL or JMP instructions, at fixed addresses.
- Shared libraries get loaded at fixed addresses within the process memory.
- Ideal for finding CALLs, JMPs.
- We can try manual pattern searching with opcodes, using a debugger…
- …or we can use msfpescan or msfelfscan.

msfpescan, msfelfscan

- Utilities to scan binaries (executables or shared libraries).
- Support for ELF and PE binaries.
- Uses metasploit's built-in disassemblers.
- Can find CALLs, JMPs, or POP/POP/RET instruction sets.
- Can be used to find instruction groups specified by regular expressions.

msfpescan’ning Windows DLLs

- If we need to search for a jump to ESI:

```
~/framework$ ./msfpescan -f windlls/USER32.DLL -j esi
0x77e11c46     call esi
0x77e121b7     call esi
0x77e121c5     call esi
0x77e1222a     call esi
:         :    :     :
0x77e6ca97     jmp esi
```

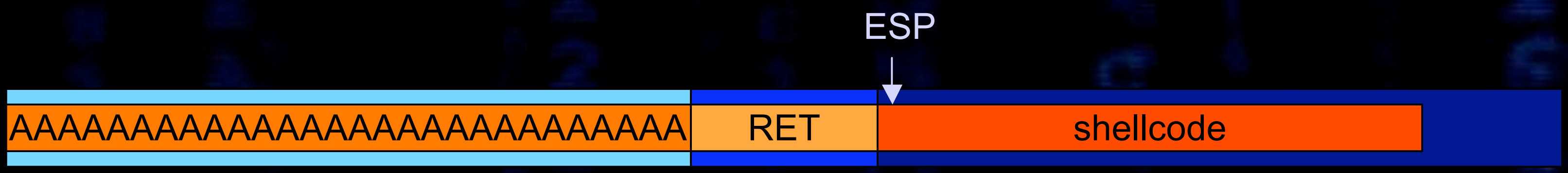
- We can point EIP to any of these values…
- …and it will then execute a JMP/CALL ESI

Candidate binaries

- First, search the executing binary itself.
  - Independent of Kernel, Service Packs, libs.
- Second, search shared libraries or DLLs included with the software itself. (e.g. in_mp3.dll for Winamp)
- Last, search default shared libraries that get included from the OS:
  - e.g. KERNEL32.DLL, libc.so, etc.
  - Makes the exploit OS kernel, SP specific.

Case Study - peercast HTTP overflow

- 1000 byte payload.
- first 780 bytes can be AAAA's.
- Bytes 781-784 shall contain an address which will go into EIP.
- Bytes 785 onwards contain shellcode.



© Saumil Shah

A little about shellcode

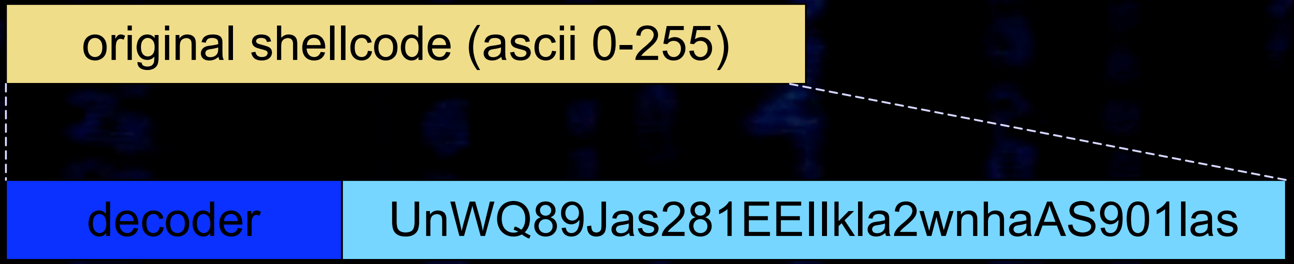
- Types of shellcode:
  - Bind shell
  - Exec command
  - Reverse shell
  - Staged shell, etc.
- Advanced techniques:
  - Meterpreter
  - Uploading and running DLLs “in-process”
  - …etc.

Payload Encoders

- Payload encoders create encoded shellcode, which meets certain criteria.
- e.g. Alpha2 generates resultant shellcode which is only alphanumeric.
- Allows us to bypass any protocol parsing mechanisms / byte filters.
- An extra “decoder” is added to the beginning of the shellcode.
  - size may increase.

Payload Encoders

- Example: Alpha2 encoding


original shellcode (ascii 0-255)
decoder
UnWQ89Jas281EEIIkla2wnhaAS901las

- Transforms raw payload into alphanumeric only shellcode.
- Decoder decodes the payload “in-memory”.

Payload Encoders

- Metasploit offers many types of encoders.
- Work around protocol parsing
  - e.g. avoid CR, LF, NULL
  - toupper(), tolower(), etc.
- Defeat IDS
  - Polymorphic Shellcode
  - Shikata Ga Nai

Exploiting Exception Handling

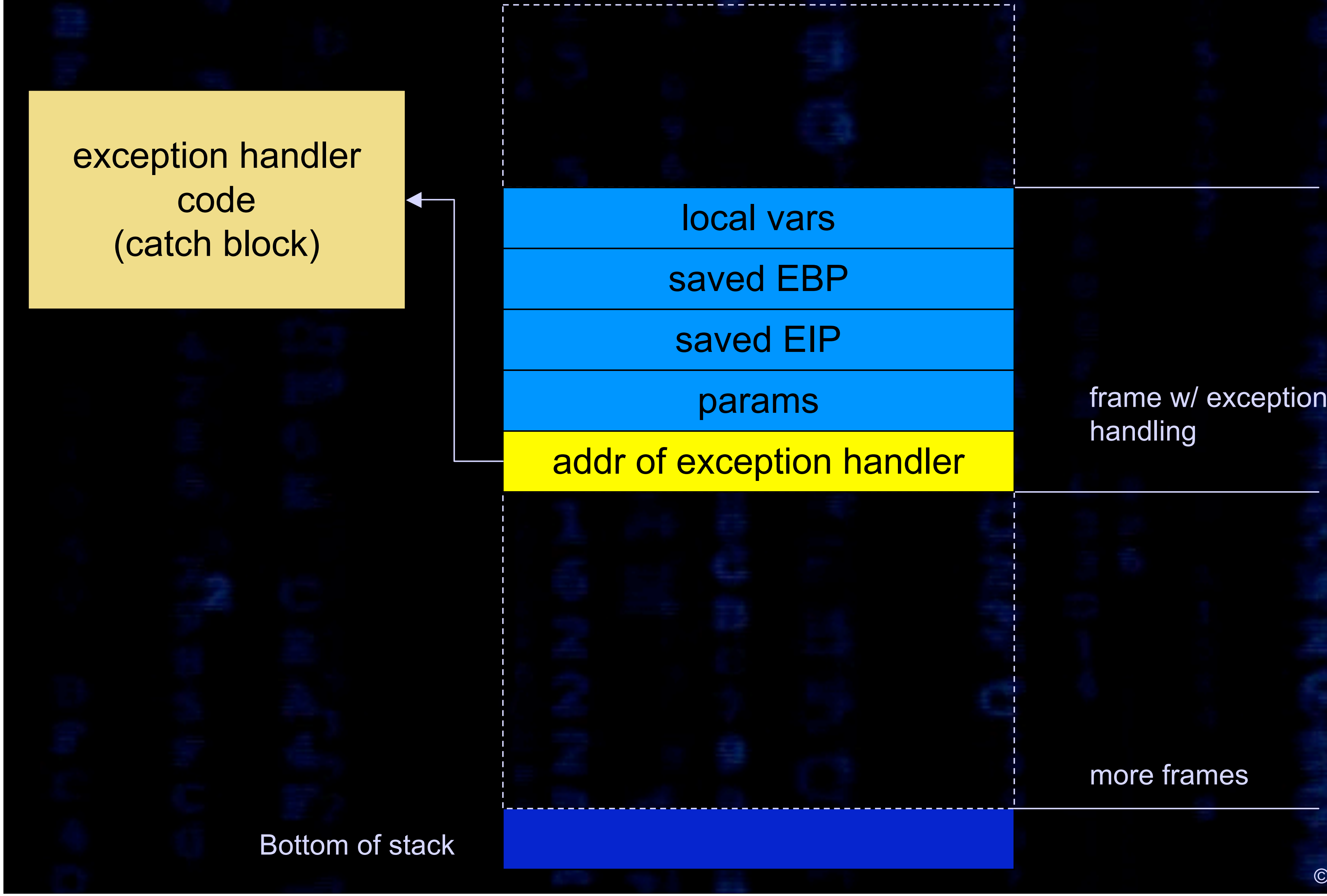
- Try / catch block

```
try {
    :         code that may throw
    :         an exception.
}
catch {
    :         attempt to recover from
    :         the exception gracefully.
}
```

- Pointer to the exception handling code also saved on the stack, for each code block.

© Saumil Shah

Exception handling … implementation


exception handler code (catch block)

local vars

saved EBP

saved EIP

params

addr of exception handler

frame w/ exception handling

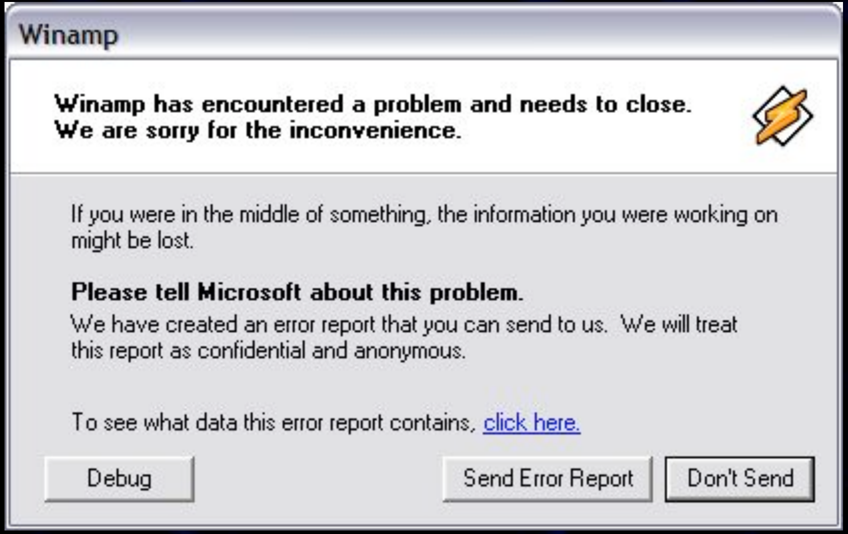
more frames

Bottom of stack

© Saumil Shah

Windows SEH

- SEH - Structured Exception Handler
- Windows pops up a dialog box:



Winamp

Winamp has encountered a problem and needs to close. We are sorry for the inconvenience.

If you were in the middle of something, the information you were working on might be lost.

Please tell Microsoft about this problem.

We have created an error report that you can send to us. We will treat this report as confidential and anonymous.

To see what data this error report contains, click here.

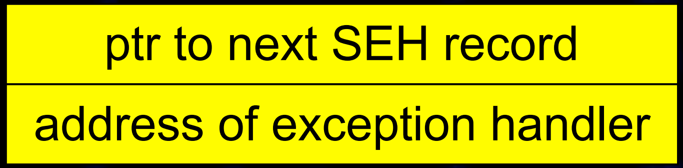
- Default handler kicking in.

Custom exception handlers

- Default SEH should be the last resort.
- Many languages including C++ provide exception handling coding features.
- Compiler generates links and calls to exception handling code in accordance with the underlying OS.
- In Windows, exception handlers form a LINKED LIST chain on the stack.

SEH Record

- Each SEH record is of 8 bytes

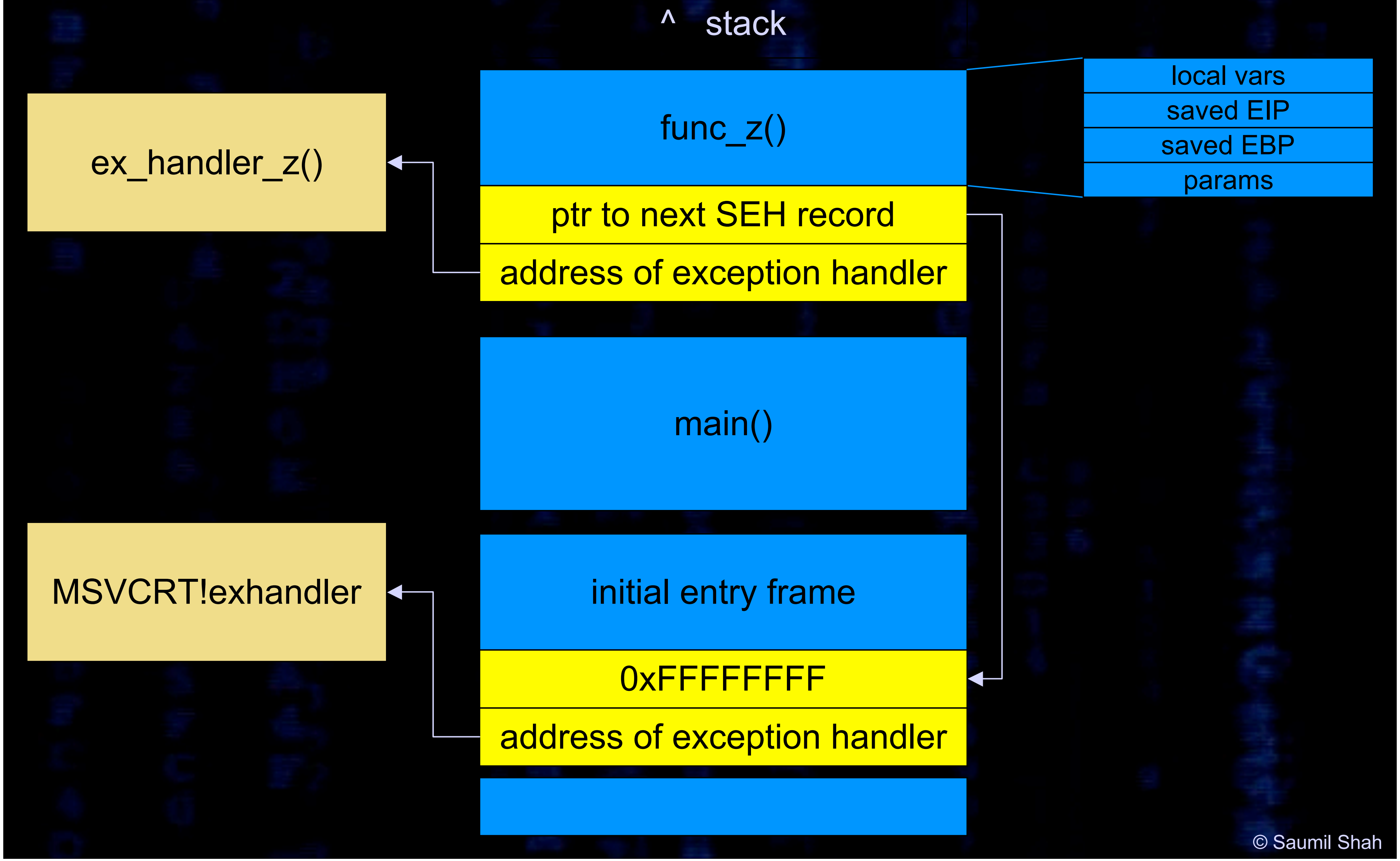


ptr to next SEH record

address of exception handler

- These SEH records are found on the stack.
- In sequence with the functions being called, interspersed among function (block) frames.
- WinDBG command - !exchain

SEH on the stack

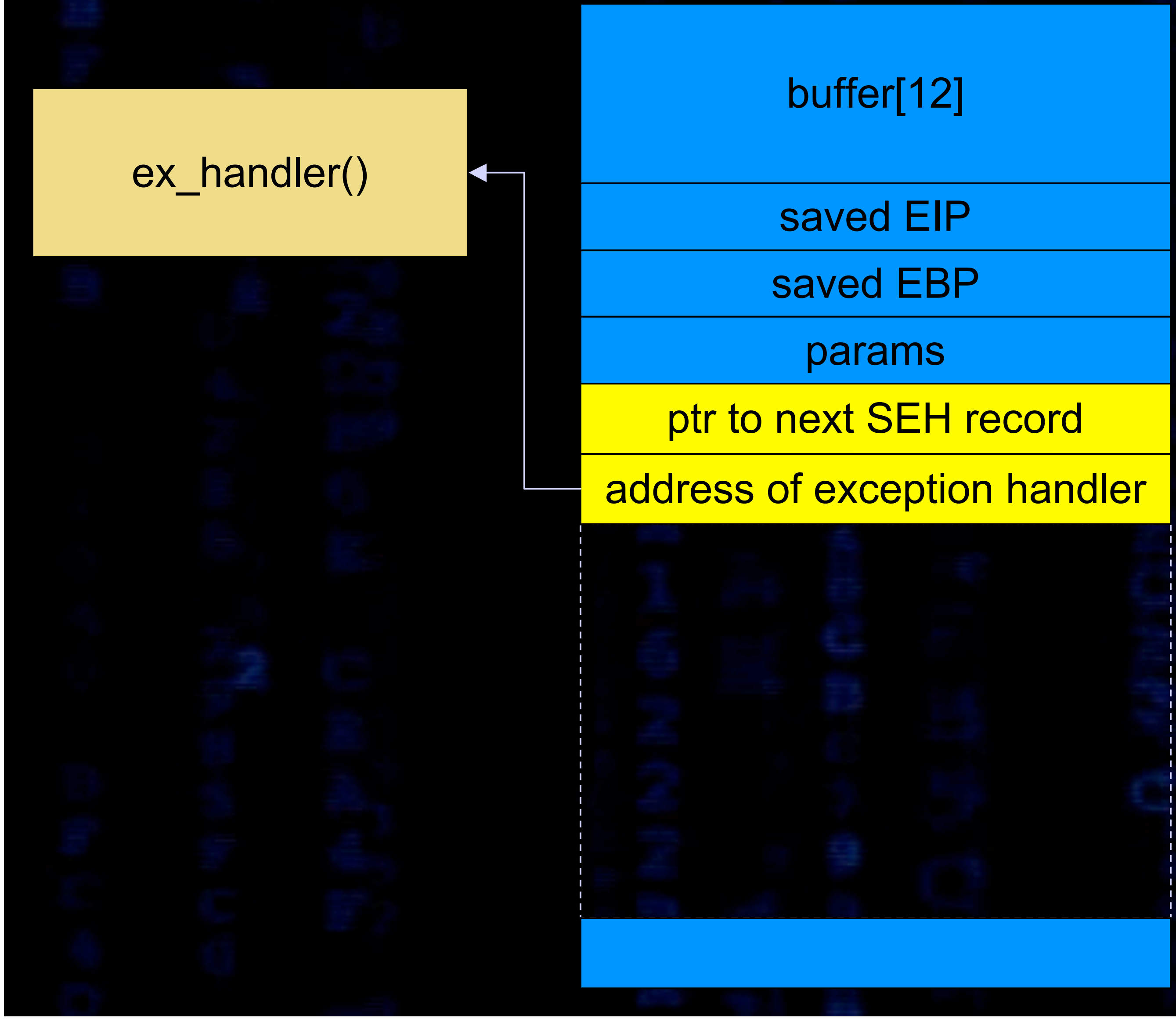


© Saumil Shah

Yet another way of getting EIP

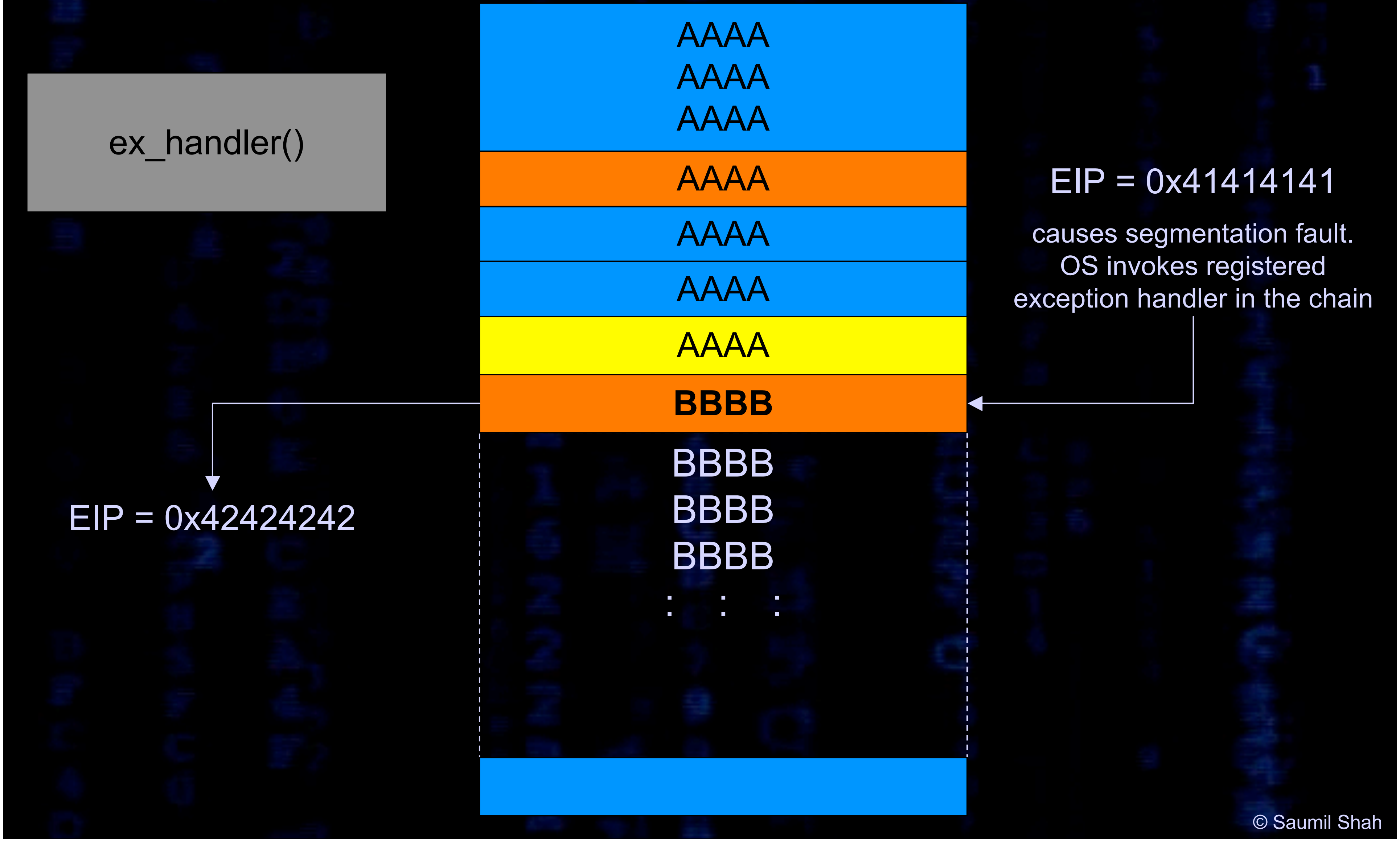
- Overwrite one of the addresses of the registered exception handlers…
- …and, make the process throw an exception!
- If no custom exception handlers are registered, overwrite the default SEH.
- Might have to travel way down the stack…
- …but in doing so, you get a long buffer!

Overwriting SEH



© Saumil Shah

Overwriting SEH

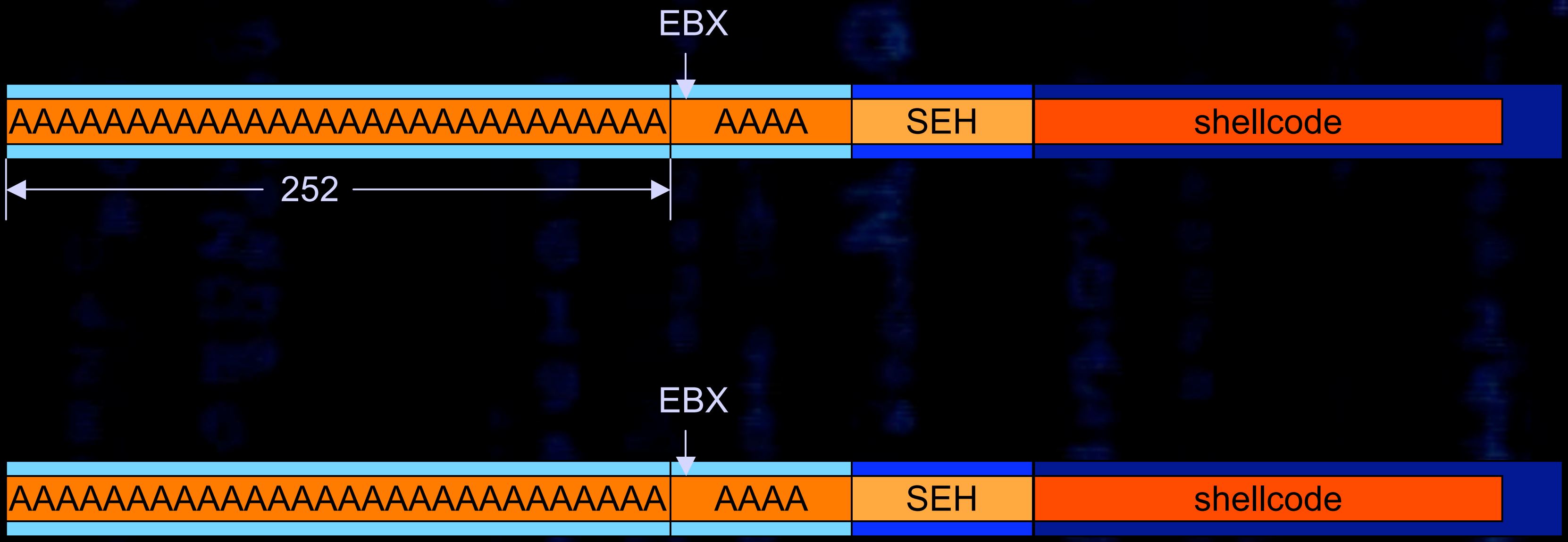


© Saumil Shah

Case study - sipXtapi CSeq overflow

- sipXtapi library - popular open source VoIP library.
- Used in many soft phones
  - AOL Triton soft phone uses sipXtapi.
- 24 byte buffer overflow in the CSeq SIP header.
- Too small for any practical shellcode.
- We can hack it up by overwriting SEH.

Putting the payload together

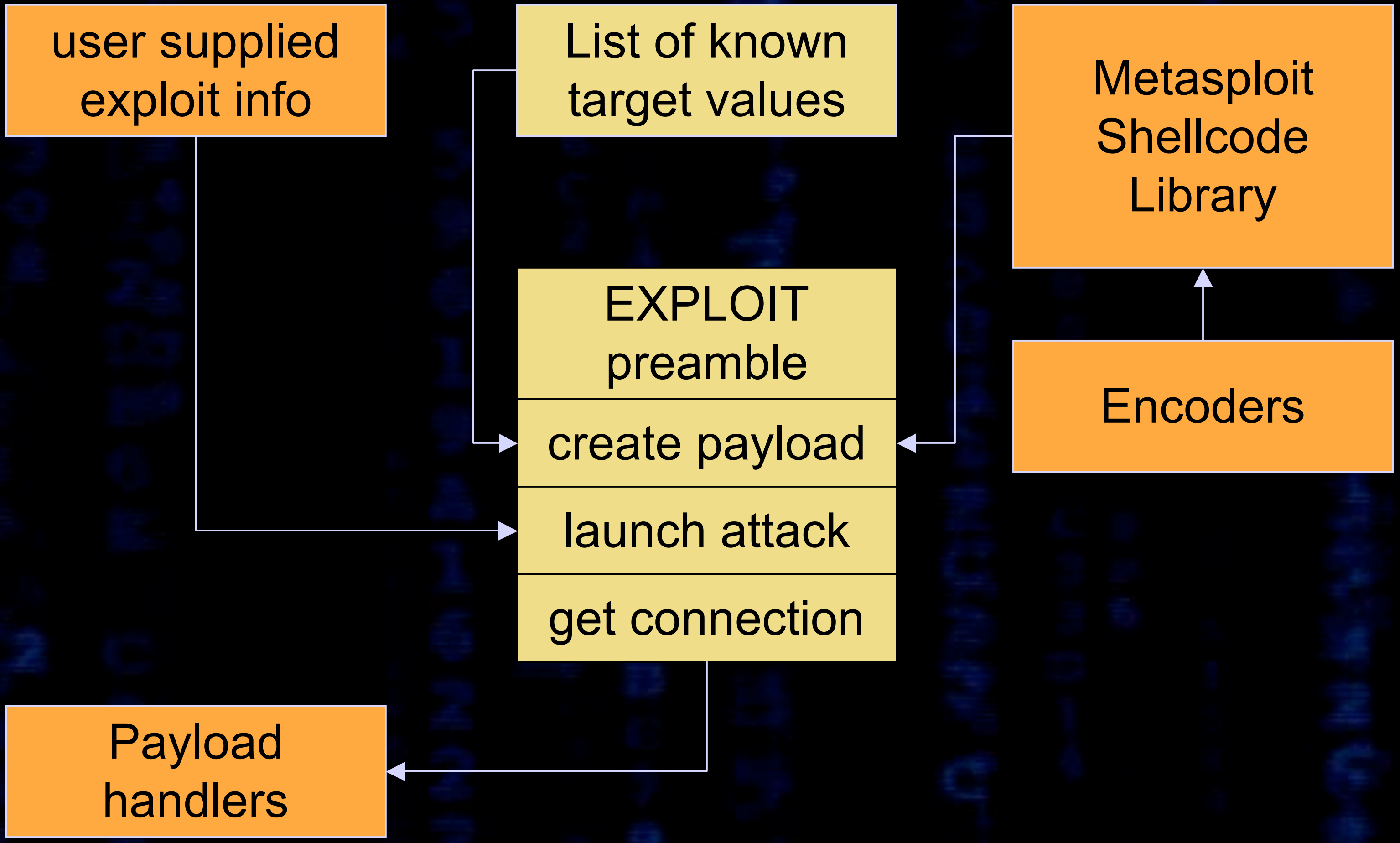


© Saumil Shah

Writing Metasploit exploit modules

- Integration within the Metasploit framework.
- Multiple target support.
- Dynamic payload selection.
- Dynamic payload encoding.
- Built-in payload handlers.
- Can use advanced payloads.
- …a highly portable, flexible and rugged exploit!

How Metasploit runs an exploit



user supplied exploit info
List of known target values
Metasploit Shellcode Library
EXPLOIT preamble
create payload
launch attack
get connection
Encoders
Payload handlers

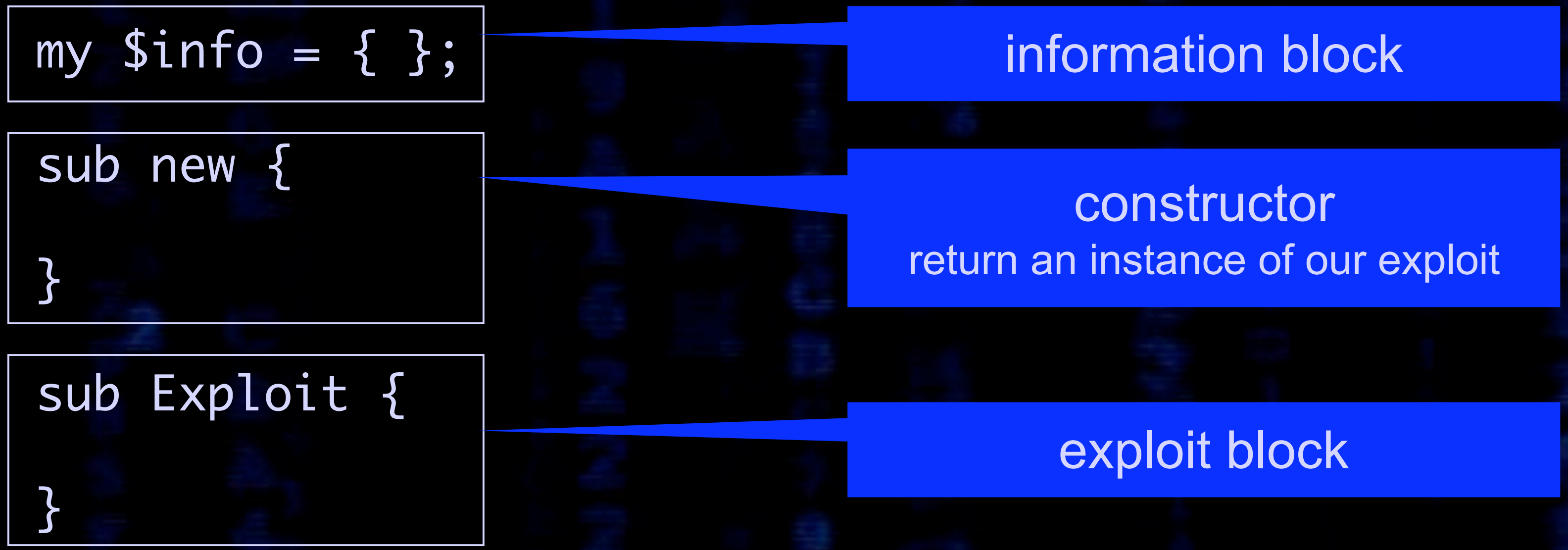
Writing a Metasploit exploit

- Perl module (2.6), Ruby module (3.0)
- Pre-existing data structures
  - %info, %advanced
- Constructor
  - sub new {…}
- Exploit code
  - sub Exploit {…}

Structure of the exploit perl module

```
package Msf::Exploit::name;
use base “Msf::Exploit”;
use strict;
use Pex::Text;

my $advanced = { };
```



```
my $info = { };

sub new {

}

sub Exploit {

}
```

information block

constructor
return an instance of our exploit

exploit block

%info

- Name
- Version
- Authors
- Arch
- OS
- Priv
- UserOpts

- Payload
- Encoder
- Refs
- DefaultTarget
- Targets
- Keys

Metasploit Pex

- Perl EXtensions.

  <metasploit_home>/lib/Pex.pm

  <metasploit_home>/lib/Pex/
- Text processing routines.
- Socket management routines.
- Protocol specific routines.
- These and more are available for us to use in our exploit code.

Pex::Text

- Encoding and Decoding (e.g. Base64)
- Pattern Generation
- Random text generation (to defeat IDS)
- Padding
- …etc

Pex::Socket

- TCP
- UDP
- SSL TCP
- Raw UDP

Pex - protocol specific utilities

- SMB
- DCE RPC
- SunRPC
- MSSQL
- …etc

Pex - miscellaneous utilities

- Pex::Utils
- Array and hash manipulation
- Bit rotates
- Read and write files
- Format String generator
- Create Win32 PE files
- Create Javascript arrays
- …a whole lot of miscellany!

metasploit_skel.pm

- A skeleton exploit module.
- Walk-through.
- Can use this skeleton to code up exploit modules.
- Place finished exploit modules in:

  <path_to_metasploit>/exploits/

Finished examples

- my_peercast.pm
- my_sipxtapi.pm

© Saumil Shah

Some command line Metasploit tools

- msfcli
  - Metasploit command line interface.
  - Can script up metasploit framework actions in a non-interactive manner.
- msfpayload
  - Generate payload with specific options.
- msfencode
  - Encode generated payload.

More command line Metasploit tools

- msfweb
  - Web interface to the Metasploit framework.
- msfupdate
  - Live update for the Metasploit framework.

New in Version 3.0

- msfd
  - Metasploit daemon, allows for client-server operation of Metasploit.
- msfopcode
  - command line interface to Metasploit's online opcode database.
- msfwx
  - a GUI interface using wxruby.

New in Version 3.0

- New payloads, new encoders.
- Ruby extension - Rex (similar to Pex)
- NASM shell.
- Back end Database support.
- …whole lot of goodies here and there.

Thank You!

Saumil Shah

saumil@saumil.net | http://net-square.com

+91 98254 31192