



# Defeating DEP, the Immunity Debugger way

Pablo Solé  
**Immunity**

# Old School Stack Overflow

Stack Memory

	Buffer	
	.	
	.	
	Saved Frame	
	Return Address	
	Func Args	
	.	
	.	
	.	
	.	
	.	

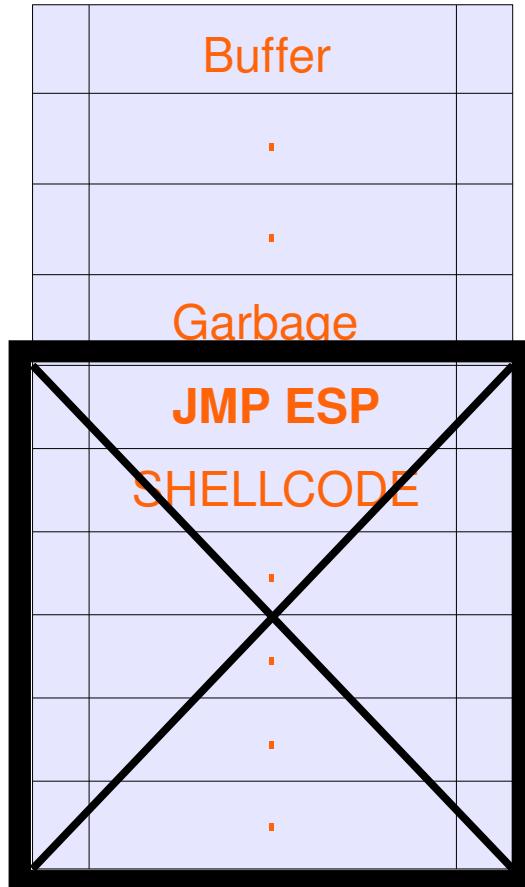
Stack Memory

	Buffer	
	.	
	.	
	Garbage	
	<b>JMP ESP</b>	
	<b>SHELLCODE</b>	
	.	
	.	
	.	
	.	



# Data Execution Prevention

Stack Memory

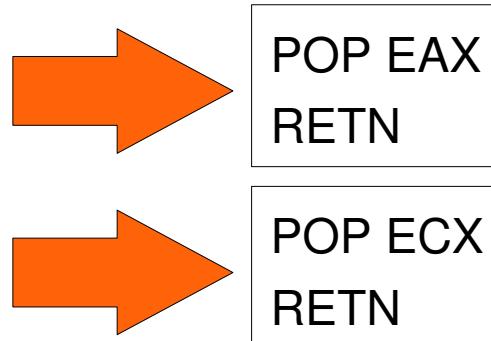


stack and heap  
are not executable anymore!

# Return to LIBC 101

Stack Memory

	Addr-POP EAX
	<b>VALUE for EAX</b>
	Addr-POP ECX
	<b>VALUE for ECX</b>
	Function Addr
	...
	...
	...
	...
	...



Let's say we want to call a function  
that takes the arguments from  
registers EAX and ECX

# Survey of the Landscape

Data Execution Prevention

vs

Immunity Debugger

What I Have

CPU/Memory Context  
Control

What I Want

EAX=alloc(0x1000, RWX)  
memcpy(EAX, shellcode)  
jmp(EAX)

What I Need

Stack String  
(that bypass filters)

# Some Previous Efforts

- EEREAP by eEye
  - made a cpu/memory emulator using process snapshots
- Return-Oriented Programming by Hovav Shacham
  - presented a Turing-complete language using pre-selected opcode sequences (gadgets)
- manual efforts
  - you can always search your own ret-to-libc pieces manually

# An Aside

I need a pop/pop/ret FTW!

EIP = [ESP + 8]

```
POP R32  
POP R32  
RETN
```

```
ADD ESP, 8  
RETN 30
```

```
XCHG EAX, ESP  
LEA EAX, [EAX+8]  
MOV EDX, [EAX]  
XCHG EAX, ESP  
RETN
```

DEPLib.seteip(['ESP','+',8])

Hacker's needs  
(Lazy Level)

Mind (Theoretical Level)

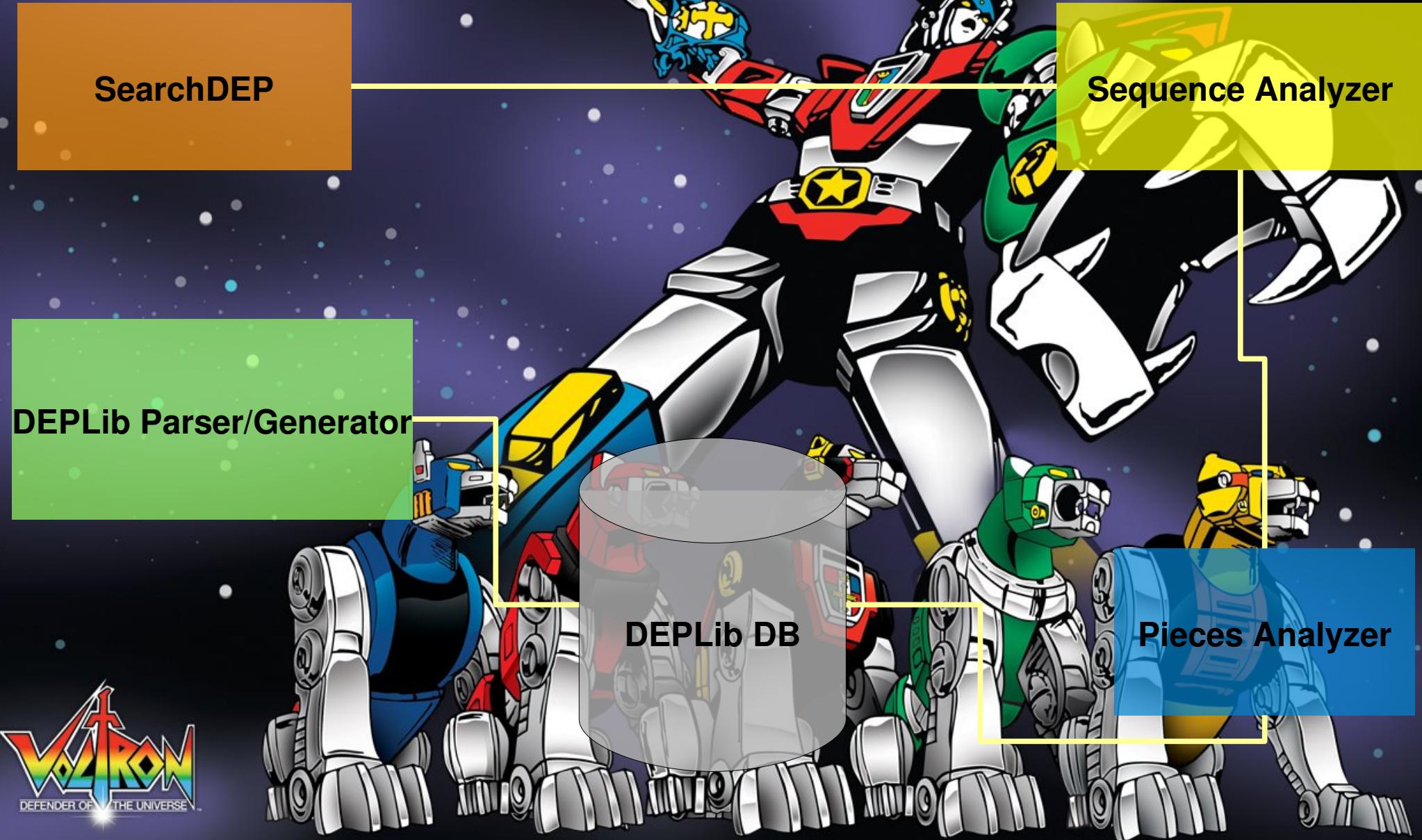
Reality (Binary level)

Magic! (DEPLib level)

# Real World Problems

- Win32, not Linux
- Cannot be limited to libc (or any particular library)
- Optimize for filter bypassing and size
- Simplistic language
- It should find the necessary sequences automatically

# Many pieces combining = VOLTRON



# SearchDEP

- Search RETN opcodes (0xC2 or 0xC3) in the entire DLL memory
- Disassemble backward until it finds an unsupported/invalid opcode
- Generate all possible disassemblies (move a byte and magic can occur)
- Finally, it returns lists of opcodes for each RETN-ended sequence

# SearchDEP Example

## Binary Data

7C91990D	66	83	26	00	66	83	66	ff&.fff
7C919915	02	00	83	66	04	00	5E	5D .ff.^]
7C91991D	C2	04	00					Â.

## Possible Disassemblies

7C91991A	005E 5D	ADD BYTE PTR DS:[ESI+5D],BL
7C91991D	C2 0400	RETN 4
7C919919	04 00	ADD AL,0
7C91991B	5E	POP ESI
7C91991C	5D	POP EBP
7C91991D	C2 0400	RETN 4
7C91990E	668326 00	AND WORD PTR DS:[ESI],0
7C919912	668366 02 00	AND WORD PTR DS:[ESI+2],0
7C919917	8366 04 00	AND DWORD PTR DS:[ESI+4],0
7C91991B	5E	POP ESI
7C91991C	5D	POP EBP
7C91991D	C2 0400	RETN 4

# Sequence Analyzer

- Emulate each instruction
- Generate a resulting CPU/Memory context
- Support interactions between CPU and Memory
- Solve modulo  $2^{8/16/32}$  arithmetic
  - $a \text{ xor } a = 0$  /  $a \text{ and } 0 = 0$
  - $a \text{ xor } !a = \text{all-ones}$
  - $(a >> 16) \text{ and } !(a >> 16) = 0$
- Support abstract memory addressing
  - MOV EAX, DWORD PTR DS:[EDX]  
(supposing we don't know EDX value)

# CPU/Memory Context

```
7C9EA06C Analyzing: MOV DWORD PTR DS:[EAX],0
7C9EA072 Analyzing: MOV EBX,DWORD PTR DS:[EAX]
7C9EA074 Analyzing: PUSH EBX
memory
[[['='], ['unk'], 'ESP'], 32, 0], ['-'], ['con', 3L], 32, 0]]:[[['='], ['con', 0L], 8, 0]]
[[['='], ['unk'], 'EAX'], 32, 0], ['+'], ['con', 1], 32, 0]]:[[['='], ['con', 0L], 8, 0]]
[[['='], ['unk'], 'EAX'], 32, 0]]:[[['='], ['con', 0L], 8, 0]]
[[['='], ['unk'], 'ESP'], 32, 0], ['-'], ['con', 1L], 32, 0]]:[[['='], ['con', 0L], 8, 0]]
[[['='], ['unk'], 'ESP'], 32, 0], ['-'], ['con', 2L], 32, 0]]:[[['='], ['con', 0L], 8, 0]]
[[['='], ['unk'], 'ESP'], 32, 0], ['-'], ['con', 4], 32, 0]]:[[['='], ['con', 0L], 8, 0]]
[[['='], ['unk'], 'EAX'], 32, 0], ['+'], ['con', 3L], 32, 0]]:[[['='], ['con', 0L], 8, 0]]
[[['='], ['unk'], 'EAX'], 32, 0], ['+'], ['con', 2L], 32, 0]]:[[['='], ['con', 0L], 8, 0]]
registers
ESP:[[['='], ['unk'], 'ESP'], 32, 0], ['-'], ['con', 4], 32, 0]]
EDI:[[['='], ['unk'], 'EDI'], 32, 0]]
EAX:[[['='], ['unk'], 'EAX'], 32, 0]]
EDX:[[['='], ['unk'], 'EDX'], 32, 0]]  
EBX:[[['='], ['con', 0L], 32, 0]]
ESI:[[['='], ['unk'], 'ESI'], 32, 0]]
ECX:[[[=], ['unk'], 'ECX'], 32, 0]]
EIP:[[[=], ['unk'], 'EIP'], 32, 0]]
RETM offset:0x0
elapsed time in secs:0
```

!sequenceanalyzer

# Pieces Analyzer

- Summarize the consequences of executing a sequence over our running context
- Make hashes of each CPU/Memory change
- Calculate a complexity value of each piece
- We don't need to deal with instructions anymore, just CPU/Memory state

# Consequences

- Is a change in the CPU/Memory context
- Each piece generate a set of consequences
- We can search in our DB for these consequences using the hashes

**EBX: [ [ '=' , [ 'con' , 0L] , 32L, 0L ] ] :C59756C0**

Reg	Value	CRC32
-----	-------	-------

# Piece Complexity Index

- How complex is this piece?
  - how many consequences does it have?
  - how many memory operations does it have?
  - how much has the stack pointer moved?

```
MOV EDI,EDI  
RETN
```

```
MOV EDI,EAX  
POP EAX  
POP EBX  
RETN 4
```

```
MOV EAX,[EBX]  
POP ECX  
RETN 0C
```

```
XOR EDI,[EBX+ECX*4]  
MOV [EDI],EAX  
XOR EAX,EAX  
POP ECX  
RETN 30
```

COMPLEXITY



# ID Database

- Store all module's pieces along with necessary information to replay the sequence
- Using the consequence hash we can find suitable pieces quick and easy
- And we always get the simpler piece that does the job thanks to the complexity index

# The DB

```
SELECT * FROM pieces WHERE piece_id IN  
(SELECT piece_id FROM consequences WHERE consequence_hash = "32D7A775")  
ORDER BY piece_complexity  
LIMIT 5
```



Run SQL    Last Error: not an error

piece_id	piece_complexity	module_id	size	module_offset	piece_dump	piece_properties
37	1	1	2	5921	BLOB (Size: 1077)	['LoadStack', 'LoadReg']
5	1	1	2	4626	BLOB (Size: 1077)	['LoadStack', 'LoadReg']
4	1	1	3	4624	BLOB (Size: 1077)	['LoadStack', 'LoadReg']
2	1	1	2	4622	BLOB (Size: 1077)	['LoadStack', 'LoadReg']
1	1	1	3	4620	BLOB (Size: 1077)	['LoadStack', 'LoadReg']

# Data Data everywhere

- NTDLL generates ~10,000 pieces
- Many pieces have the same consequences  
(but we need them to bypass filters)
- Processing an average sized library takes 4 hrs
- SQL allows us to make arbitrary complex queries
  - Find pieces with the same address over a set of modules (universal addresses)
  - Use only application specific addresses (independent from the OS)

# DEPLib Parser

- Track register use to avoid undesirable overwriting of already settled values (du chains)
- It supports variables
- It does register reusing over non-overlapped variables
- Calculate all possible combinations of variable to register mapping

# About the Language

- Almost assembler:
  - mov, sub, add
  - xor, and, or
  - shl, shr, rol, ror
  - call (using fixed or dynamic stack arguments)
- And some useful additions:
  - jump to your shellcode
  - find your stack stream
    - `parser.findbuffer(['EAX','+',8])`  
means: EIP=[EAX+8], ESP=EAX+12
  - find the stack stream end

# DEPLib Parser Example

```
- def test(self):  
    imm = immlib.Debugger()  
    parser = DEPLibParser(imm)  
  
    parser.mov("EDX", 0x00040000) #f1Options (HEAP_CREATE_ENABLE_EXECUTE)  
    parser.mov("ECX", 0x00001000) #dwInitialSize  
    parser.mov("EAX", 0)          #dwMaximumSize  
    parser.call_args("KERNEL32.HeapCreate", 3)
```

# Transformation...

```
reg:EAX, defines:[2L, 3L]
reg:EDX, defines:[0L]
reg:ECX, defines:[1L]
reg:EAX, uses:[3L]
reg:EDX, uses:[3L]
reg:ECX, uses:[3L]
reg:EAX, frees:[3L]
reg:EDX, frees:[3L]
reg:ECX, frees:[3L]
reg:EAX, needed:[]
reg:EDX, needed:[1L, 2L]
reg:ECX, needed:[2L]
cmdpos: 0, cmd:[('MOV', [('reg', 'EDX'), ('const', 262144L)])]
regs:[], vars:[], defregs:['EDX'], defvars:[]

cmdpos: 1, cmd:[('MOV', [('reg', 'ECX'), ('const', 4096L)])]
regs:['EDX'], vars:[], defregs:['ECX'], defvars:[]

cmdpos: 2, cmd:[('MOV', [('reg', 'EAX'), ('const', 0L)])]
regs:['EDX', 'ECX'], vars:[], defregs:['EAX'], defvars:[]

cmdpos: 3, cmd:[('CALL_ARGS', [('const', 2088840262L), 3L, ['EAX', 'ECX', 'EDX'], False])]
regs:[], vars:[], defregs:['EAX'], defvars:[]
```

# DEPLib Generator

- Searches in the database for pieces that generate our desired consequences (we don't use hand-selected addresses, all is done dynamically)
- Checks that each piece satisfies a set of preconditions:
  - undesired memory writing/reading
  - undesired register overwriting
  - piece effective address bypass chars filtering
- Creates a stack sequence that we need and fills the blanks with good chars

# DEPLib Parameters

- To start generating our stack stream, we need the following information from the user:
  - a DEPLib Parser instance
  - a list of allowed modules to get the pieces from
    - OS specific?
    - Application specific?
    - Universal addresses?
  - a list of memory addresses where we can read or write (optional)
  - a list of bad chars (to bypass chars filtering)

# Some tricks...

- We need to support loading of arbitrary values to registers (even if they have bad chars)

```
deplib.mov('EAX', 0x00040000)
```



DEPLib Language Level

```
def mov(reg, value):  
    if value has badchars:  
        PopPopSubTrick(reg, value)  
    else:  
        PopTrick(reg, value)  
  
...  
  
def PopPopSubTrick(reg, value):  
    (val1, val2) = findSubValues(value)  
    mov(reg, val1)  
    reg2 = findFreeReg()  
    mov(reg2, val2)  
    sub(reg, reg2)
```



DEPLib Logic Level

# Some tricks...

```
MOV EAX, 0xFFFFFFFF  
MOV EDX, 0xFFFFBFFFFF  
SUB EAX, EDX
```

→ Assembler Level

## Return Programming Level

### Stack Memory

77F21564
FFFFFFFFF
77F33A40
FFFBFxFFFF
77F31293
...
...
...

77F21564 58 POP EAX
77F21565 C3 RETN
77F33A40 5A POP EDX
77F33A41 C3 RETN
77F31293 2BC2 SUB EAX,EDX
77F31295 C3 RETN

# Metrics

- We've obtained a stack stream that successfully and reliably do:
  - **HeapCreate** with the Executable Page option
  - **Allocate** a chunk in this new heap
  - **Memcpy** our shellcode
  - **Jump** to the allocated chunkIn just **280** bytes, bypassing NULL chars filters  
If you don't have badchars it's **half** that size
- The smallest stream we have created (always bypassing NULL char filtering) is **236** bytes long

# DEMO

# Future Work

- Support the entire x86 instruction set
- Interpret flags and do conditional analysis
- Support conditional execution and looping on DEPLib to create a Turing-Complete implementation
  - Not just for the FUN, but to execute shellcode selectively

# The Conclusion

Automatically defeating DEP is not just an idea  
is a FACT

Thank you for your time

Contact me at:  
**pablo.sole@immunityinc.com**