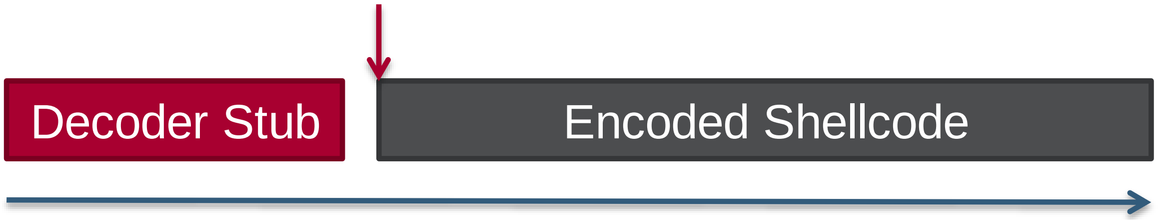
Efficient Bytecode Analysis: Linespeed Shellcode Detection

Georg Wicherski
Security Researcher

Anatomy of a Shellcode

- Little piece of Bytecode that gets jumped to in an exploit
  - Direct overwrite of EIP on the stack
  - Sprayed on the Heap and called as a function pointer
  - Allocated by small ROP payload and jumped to by last gadget
    - Minus ~~Zynamics~~ Google, they do ROPperies

- Usually some requirements because it is delivered *inline*
  - Null byte free, because it terminates a C-String
  - \r\n free, because it often is a delimiter in network protocols
  - ...



Shellcode Decoder Structure

```
jmp getpc                          ; jump to GetPC

start:                             ; GetPC 2: ebp = EIP
  pop ebp
  push 42                          ; load counter = 42
  pop ecx
  push 23                          ; load key = 23
  pop edx

decrypt:
  xor byte [ebp+ecx], dl           ; unxor one byte
  loop decrypt                     ; repeat until ecx = 0
jmp payload

getpc:
  call start                       ; GetPC 1: push EIP to stack
payload:
```

GetPC Sequences

- `call $+5, pop r32`
  - Push return address for function call onto stack
  - Use stack access to read back the return address

- `fnop, fnstenv [esp+0x0c], pop r32`
  - Use a floating point instruction, address will be stored in floating point control aread
  - Save floating point control area on stack
  - Read back the instruction address from stack

- Structured Exception Handling
  - Windows specific, trigger an exception
  - Get address of exception instruction in exception handler

Existing Detection Approaches

- Static / Statistical Approaches
  - e.g. Markov Chains for Bytecode (Alme & Elser, Caro 2009)
    - Trained with shellcode / non-shellcode data
    - Measures likelyhood of certain instructions following each other
  - Can only detect the decoder and therefore tend to be either false positive or false negative prone (weighting, training data, ...)

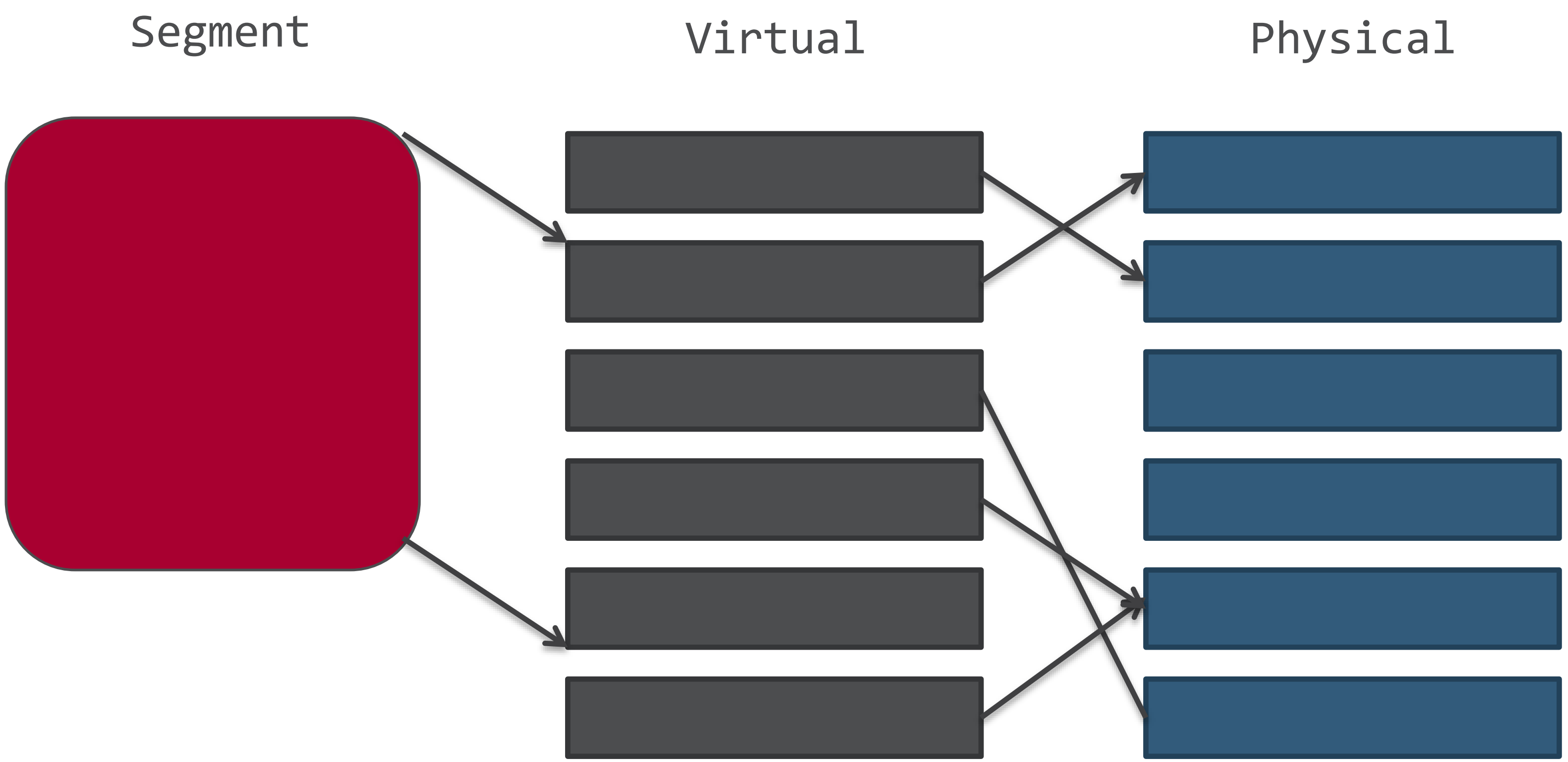
- GetPC Sequences + Backtracking + Emulation (libemu)
  - Identify possible GetPC sequences in data
  - Build up tree of possible starting locations by disassembling “backwards”
    - A problem on its own on the x86 CISC architecture
  - Software x86 emulation to weed out (the many) false positives

libscizzle

- Identification of possible GetPC sequences
  - A little less strict than libemu in terms of triggering combinations
- Brute force possible starting location around sequence
  - Efficient emulation allows this performance wise
- Use *efficient* sandboxed hardware execution for verification
  - No, this is not virtualization, no VT involved
  - Yes, it is secure, so we do not get owned (trivially)

http://code.mwcollect.org/projects/libscizzle

x86 Segmentation vs. Paging

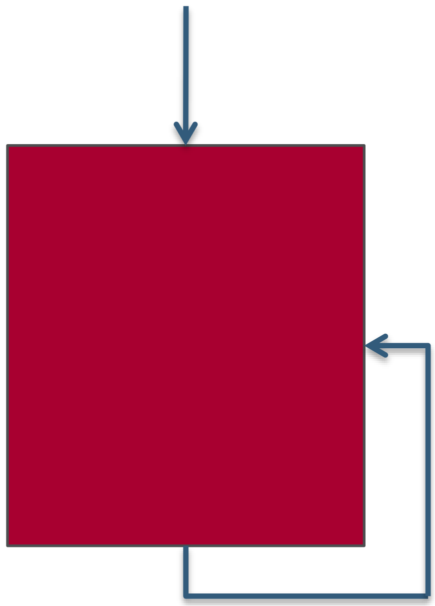


Code Execution / “Emulation”

- Disassemble guest code
  - Stop on any privileged or (potentially) execution flow modifying instruction
  - This is roughly equivalent to “basic blocks”
  - Segment register access is considered a privileged instruction ;)

- Execute one basic block at a time within the guest segment
- Emulate all other instructions
  - Conditional jumps, calls, ...
  - Abort analysis on any privileged instructions

- Exception: backwards short jumps



McAfee

Evaluation: Performance

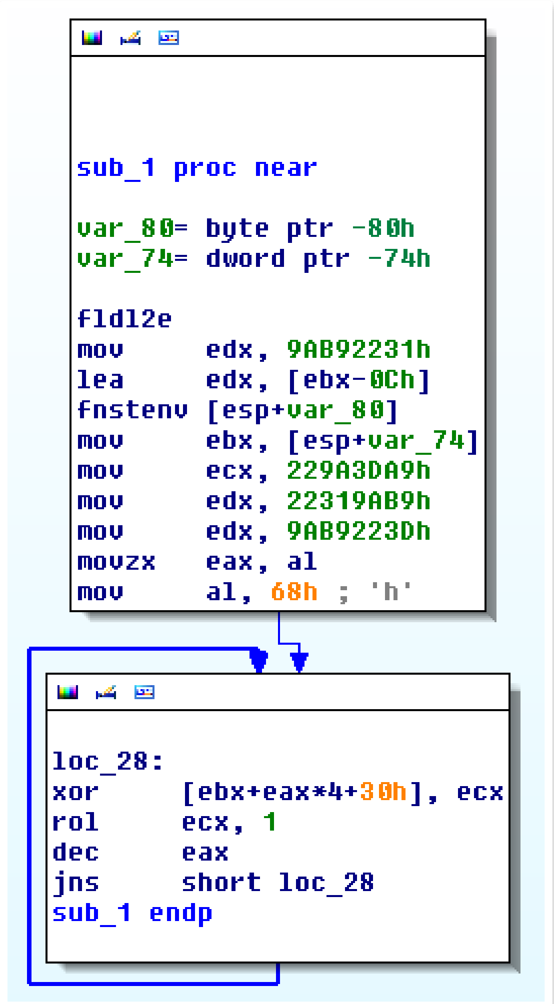
```
$ ./libscizzle-test < urandom.bin
[*] Filtering / scanning over 32.0 MiB of data took 105 ms.
[*] Verifying 700 shellcode candidate offsets...
[*] Verification over 32.0 MiB of data took 217 ms.
[*] Everything over 32.0 MiB of data took 322 ms.
```

- 99.38 Mib / sec, 795 MiB / sec on my presentation laptop, single core
- About 1000x faster than libemu, *a lot faster* than Markov Chains

- This is fast enough to do it inline at GigaBit speed on a commodity server, think IPS
- Real world data has usually better properties than purely random data

Evaluation: Success Rate

- False Positives: none.
  - If it is detected, it resembles valid shellcode
  - Random data might resemble valid shellcode but this is a philosophical problem then, highly unlikely.
- False Negatives: none so far
  - Tested on a lot of public shellcodes (tricky Metasploit ones, egghunters)
  - Used during CTFs for *testing libscizzle*, detected everything
    - DefCon, ruCTFe, ...
- Manual evasion possible



Questions?

Thanks for your attention!