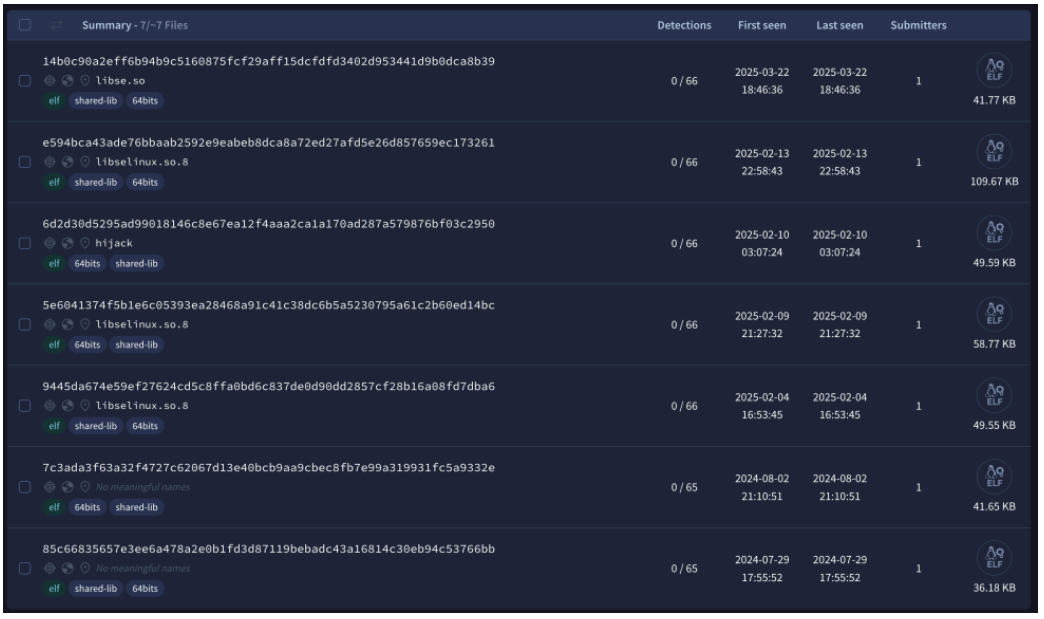
Plague: A Newly Discovered PAM-Based Backdoor for Linux

By Pierre-Henri Pezier

Published: 2026-03-30 · Archived: 2026-04-05 20:55:22 UTC

As part of our ongoing threat hunting efforts, we identified a stealthy Linux backdoor that appears to have gone publicly unnoticed so far. We named it *Plague*. The implant is built as a malicious PAM (Pluggable Authentication Module), enabling attackers to silently bypass system authentication and gain persistent SSH access.

What caught our attention: although several variants of this backdoor have been uploaded to VirusTotal over the past year, not a single antivirus engine flags them as malicious (see screenshot). To our knowledge, there are no public reports or detection rules available for this threat, suggesting that it has quietly evaded detection across multiple environments.



VirusTotal submissions of Plague samples – 0/66 detections

Plague integrates deeply into the authentication stack, survives system updates, and leaves almost no forensic traces. Combined with layered obfuscation and environment tampering, this makes it exceptionally hard to detect using traditional tools.

Its ability to persist over long periods without raising suspicion highlights the danger of backdoors targeting foundational system components like PAM. Similar threats have been described in Stealth in 100 Lines: Analyzing PAM Backdoors in Linux, underlining the broader relevance of this attack vector.

This case reinforces the importance of proactive detection through YARA-based hunting and behavioral analysis – especially for implants that operate silently at the core of Linux systems.

Threat Landscape

The presence of multiple samples, compiled over a long period and across different environments, demonstrates active development and adaptation by the threat actors.

SHA-256	Size	Filename	First submission	Submit from	Com artif
`85c66835657e3ee6a478a2e0b1fd3d87119bebadc43a16814c30eb94c53766bb`	36.18 KB	`libselinux.so.8`	2024-07-29 17:55:52	🇺🇸 USA	GCC 10.2 2021

SHA-256	Size	Filename	First submission	Submit from	Com artif
`7c3ada3f63a32f4727c62067d13e40bcb9aa9cbec8fb7e99a319931fc5a9332e`	41.65 KB	`libselinux.so.8`	2024-08-02 21:10:51	🇺🇸 USA	GCC 10.2 2021
`9445da674e59ef27624cd5c8ffa0bd6c837de0d90dd2857cf28b16a08fd7dba6`	49.55 KB	`libselinux.so.8`	2025-02-04 16:53:45	🇺🇸 USA	GCC 13.3 6ubu 13.3
`5e6041374f5b1e6c05393ea28468a91c41c38dc6b5a5230795a61c2b60ed14bc`	58.77 KB	`libselinux.so.8`	2025-02-09 21:27:32	🇺🇸 USA	GCC 13.3 6ubu 13.3
`6d2d30d5295ad99018146c8e67ea12f4aaa2ca1a170ad287a579876bf03c2950`	49.59 KB	`hijack`	2025-02-10 03:07:24	🇨🇳 CHINA	GCC 9.4.0 1ubu 9.4.0
`e594bca43ade76bbaab2592e9eabeb8dca8a72ed27afd5e26d857659ec173261`	109.67 KB	`libselinux.so.8`	2025-02-13 22:58:43 UTC	🇺🇸 USA	strip
`14b0c90a2eff6b94b9c5160875fcf29aff15dcfdfd3402d953441d9b0dca8b39`	41.77 KB	`libse.so`	2025-03-22 18:46:36	🇺🇸 USA	GCC 4.8.5 (Red

The binaries still contain compiler version metadata, which is consistent with a continuously maintained backdoor that has been running for an extended period. Some binaries are not fully stripped, further supporting this observation.

Attribution for this backdoor remains unclear. However, the presence of a sample named `hijack`, which appears to be one of the earliest, may hint at the malware's origin.

The authors left a reference to the movie Hackers visible only after deobfuscation. This is printed after `pam_authenticate` and serves as a motd message:

> "Uh. Mr. The Plague, sir? I think we have a hacker."

Capabilities and Impact

Plague backdoor is equipped with several features:

- **Antidebug:** Hinders analysis and reverse engineering.
- **String obfuscation:** Protects sensitive strings and offsets, complicating detection.
- **Static password:** Allows covert access for attackers.
- **Hidden Session artifacts:** Erases traces of attacker activity.

Technical details

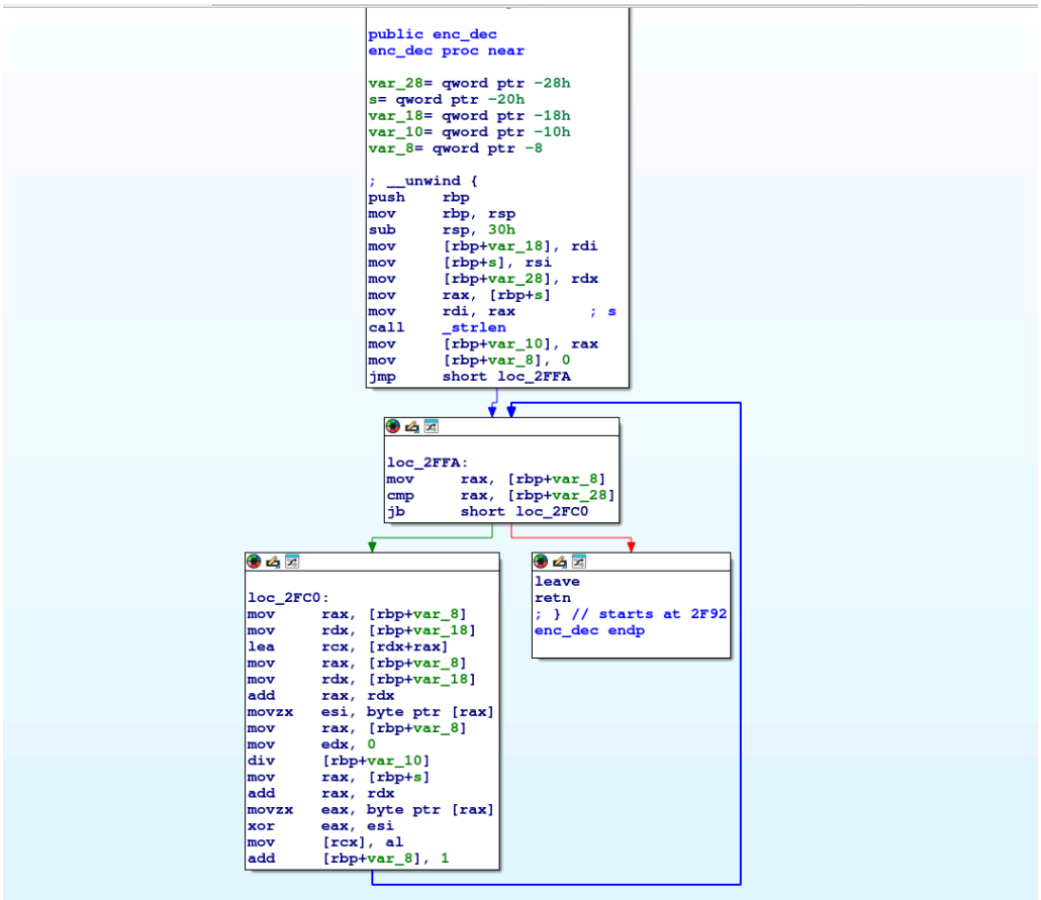
Obfuscation

The Plague backdoor employs evolving string obfuscation techniques to hinder detection and analysis. Initial samples used simple XOR-based encryption, but later versions adopted more complex methods resembling KSA and PRGA routines. The most recent variants add a DRBG layer, further complicating extraction.

These changes reflect the threat actor's ongoing efforts to evade both automated and manual analysis. The obfuscation not only hides sensitive strings but also their memory offsets, making static analysis unreliable.

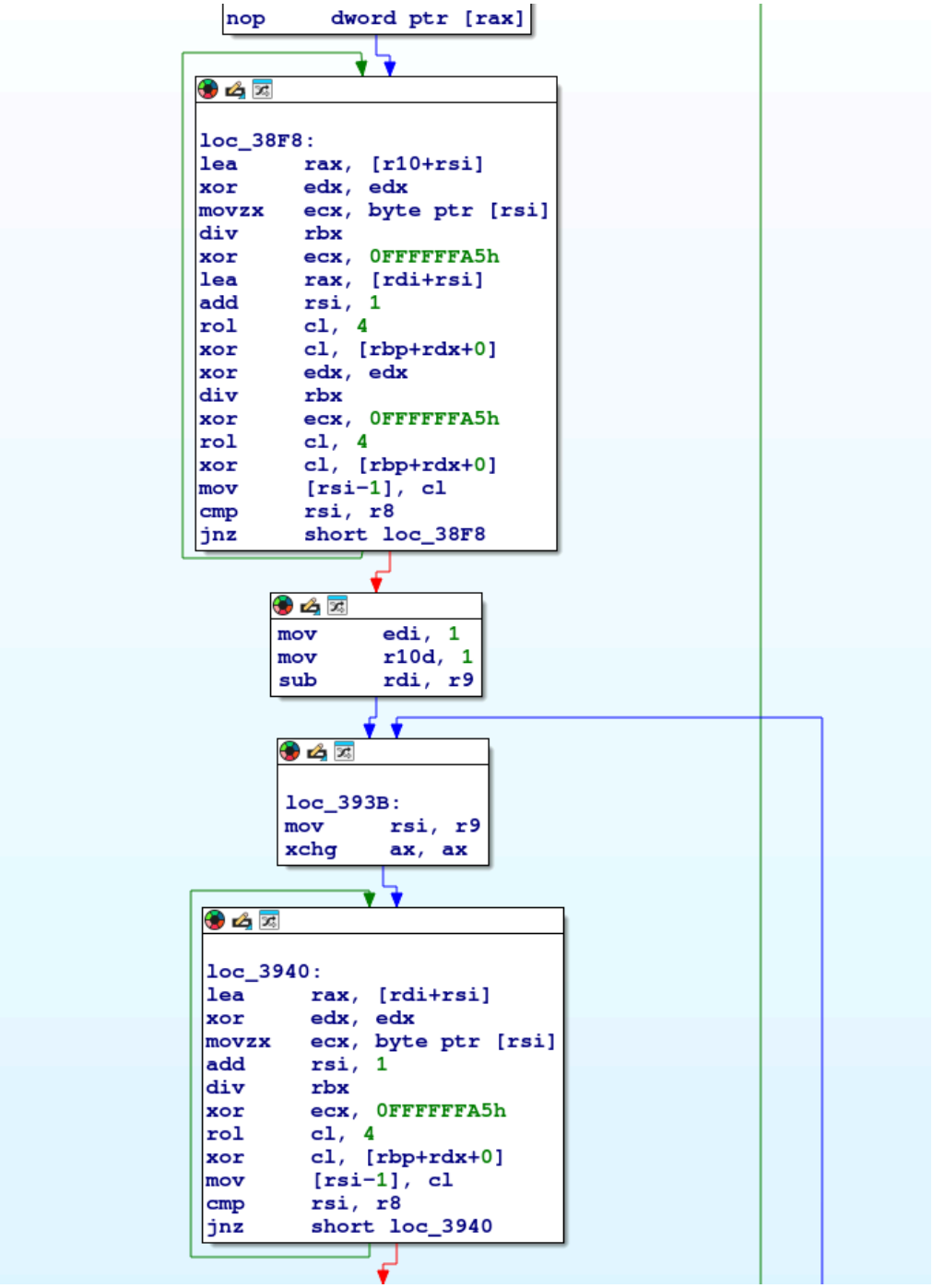
To address this, a custom string deobfuscation tool was developed using Unicorn for safe emulation within IDA Pro. This approach allows analysts to extract and annotate decrypted strings, even as the obfuscation evolves.

A decryption routine called `init_phrases` is meant to decrypt a block of data containing all the strings. When a string is needed, the `decrypt_phrase` function is called, to retreive its address which is obfuscated too. This is the first xor layer:



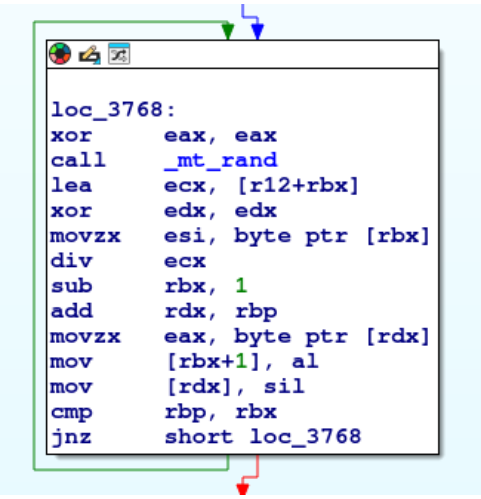
xor string decryption

Then, the custom KSA/PRGA, which acts as a layer 2:



Key derivation algorithm

And finally the third DRBG layer 3 of the obfuscation, which is not yet supported by the decryption tool.

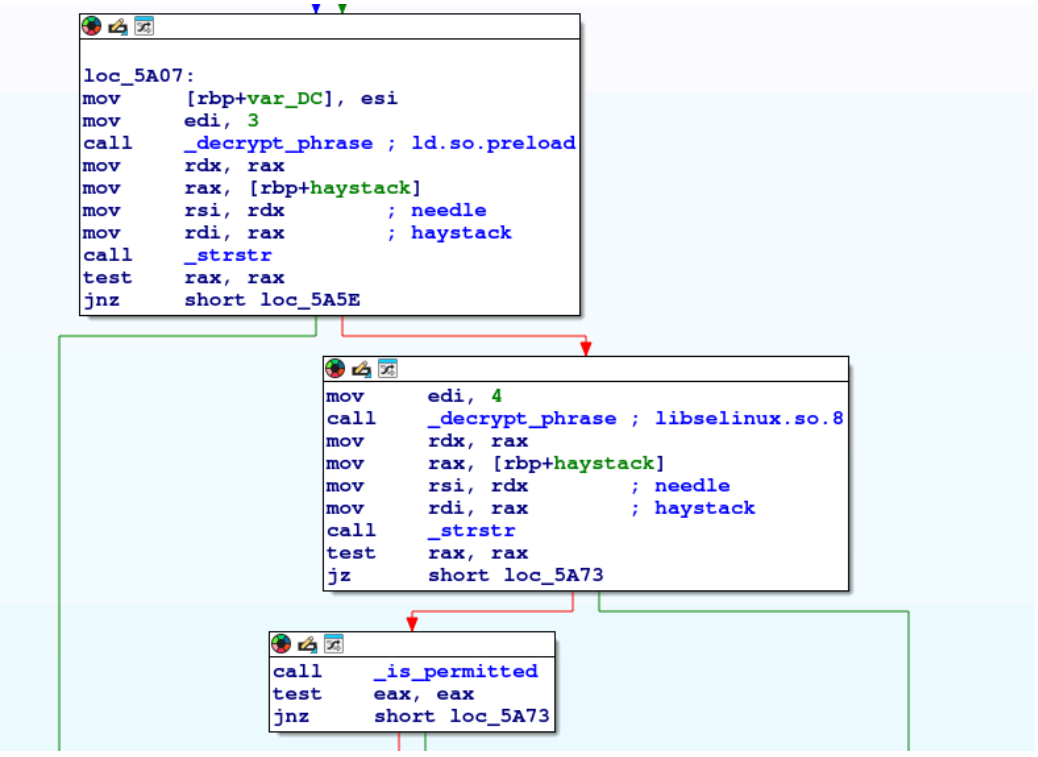


DRBG

A custom string extraction tool was developed to handle strings from a sample. Since the obfuscation algorithm has evolved and it is relatively complex to make a generic decryption tool, the extraction script has been based on unicorn to safely emulate the code from any platform on IDA Pro. The script has been attached in appendix.

Antidebug

To guarantee safe execution, the sample verifies that its actual filename is `libselinux.so.8` and that `ld.so.preload` is not present in the environment variables. This check allows the sample to evade debuggers and sandbox environments, which often rely on preload mechanisms or rename binaries during analysis.



Antidebug

Stealth

As demonstrated in the disassembly, the malware actively sanitizes the runtime environment to eliminate evidence of an SSH session. Environment variables such as `SSH_CONNECTION` and `SSH_CLIENT` are unset using `unsetenv`, while `HISTFILE` is redirected to `/dev/null` to prevent shell command logging. This operation ensures that no audit trail or login metadata is retained, effectively erasing the attacker's footprint from both interactive sessions and system history logs.

```
mov     eax, [rbp+var_1D4]
mov     esi, 0          ; fd2
mov     edi, eax        ; fd
call    _dup2
mov     eax, [rbp+var_1D4]
mov     esi, 1          ; fd2
mov     edi, eax        ; fd
call    _dup2
mov     eax, [rbp+var_1D4]
mov     esi, 2          ; fd2
mov     edi, eax        ; fd
call    _dup2
mov     edi, 0Eh
call    _decrypt_phrase ; bkr
mov     edx, 1          ; replace
lea     rsi, value      ; "1"
mov     rdi, rax        ; name
call    _setenv
mov     edi, 8
call    _decrypt_phrase ; HISTFILE=/dev/null
mov     rdi, rax        ; string
call    _putenv
mov     edi, 12h
call    _decrypt_phrase ; SSH_CONNECTION
mov     rdi, rax        ; name
call    _unsetenv
mov     edi, 13h
call    _decrypt_phrase ; SSH_CLIENT
mov     rdi, rax        ; name
call    _unsetenv
mov     edi, 5
call    _decrypt_phrase ; .-clr
mov     rbx, rax
mov     edi, 6
call    _decrypt_phrase ; /bin/bash
mov     edx, 0
mov     rsi, rbx        ; arg
mov     rdi, rax        ; path
mov     eax, 0
call    _execl
mov     edi, 5
call    _decrypt_phrase ; .-clr
mov     rbx, rax
mov     edi, 7
call    _decrypt_phrase ; /bin/sh
mov     edx, 0
mov     rsi, rbx        ; arg
mov     rdi, rax        ; path
mov     eax, 0
call    _execl
```

Session stealth

Conclusion

The Plague backdoor represents a sophisticated and evolving threat to Linux infrastructure, exploiting core authentication mechanisms to maintain stealth and persistence. Its use of advanced obfuscation, static credentials, and environment tampering makes it particularly difficult to detect using conventional methods.

THOR is continuously improving to detect even the most stealthy implants.

Detection

Artifacts

The backdoor includes hardcoded passwords to enable covert access without user authentication. The following passwords have been extracted from various samples:

- Mvi4Odm6tld7
- IpV57KNK32Ih

- changeme

The variable `bkr=1` acts as a flag indicating whether the sample is running in a safe (non-monitored) environment.

YARA

```
rule MAL_LNX_PLAGUE_BACKDOOR_Jul25 {
   meta:
      description = "Detects Plague backdoor ELF binaries, related to PAM authentication alteration."
      reference = "Internal Research"
      author = "Pezier Pierre-Henri"
      date = "2025-07-25"
      score = 80
      hash = "14b0c90a2eff6b94b9c5160875fcf29aff15dcfdfd3402d953441d9b0dca8b39"
      hash = "7c3ada3f63a32f4727c62067d13e40bcb9aa9cbec8fb7e99a319931fc5a9332e"
   strings:
      $s1 = "decrypt_phrase"
      $s2 = "init_phrases"
   condition:
      uint32be(0) == 0x7f454c46
      and filesize < 1MB
      and all of them
}
```

Appendix

The deobfuscation tool below emulates the string decryption routine using Unicorn within IDA Pro 9 and Python >= 3.10. Decrypted strings are automatically annotated. The sample is not debugged but emulated, that guaranties safe usage on multiple platforms.

```
import binascii
from unicorn import *
from unicorn.x86_const import *
import ida_segment
import ida_bytes
import ida_funcs
import ida_nalt
import idc
import idautils

class Runner:
    # Constants for the emulated stack
    STACK_ADDR = 0x0FF00000
    STACK_SIZE = 0x10000

    def __init__(self):
        # Initialize Unicorn in 64-bit x86 mode
        self.mu = Uc(UC_ARCH_X86, UC_MODE_64)
        self.hook_list = {}

        # Determine the range of memory to map based on IDA segments
        self.low_addr = min(ida_segment.getnseg(i).start_ea for i in range(ida_segment.get_segm_qty()))
        self.length = max(self.align(ida_segment.getnseg(i).end_ea - self.low_addr) for i in range(ida_segment

        # Map binary memory and stack into Unicorn
        self.mu.mem_map(self.low_addr, self.length)
        print("Mapped binary memory:", hex(self.low_addr), "size:", hex(self.length))
        self.mu.mem_map(self.STACK_ADDR, self.STACK_SIZE)

        # Copy IDA's segment bytes into Unicorn memory
        for i in range(ida_segment.get_segm_qty()):
            seg = ida_segment.getnseg(i)
            data = ida_bytes.get_bytes(seg.start_ea, seg.end_ea - seg.start_ea)
```

```
            if data:
                self.mu.mem_write(seg.start_ea, data)

        # Load imported function thunks to hook
        for addr, name in self.get_imports():
            self.hook_list[addr] = name

    @staticmethod
    def align(size, alignment=0x1000):
        # Align size to nearest page boundary
        return (size + alignment - 1) & ~(alignment - 1)

    def exec_func(self, func_name: str | int) -> int:
        # Resolve function address from name or address
        if isinstance(func_name, str):
            func = ida_funcs.get_func(idc.get_name_ea_simple(func_name))
        else:
            func = ida_funcs.get_func(func_name)

        start_offset = func.start_ea

        # Set up the stack with a fake return address (0x0)
        rsp = self.STACK_ADDR + self.STACK_SIZE // 2 - 8
        self.mu.mem_write(rsp, (0).to_bytes(8, 'little'))  # push 0
        self.mu.reg_write(UC_X86_REG_RSP, rsp)

        # Install instruction hook
        self.mu.hook_add(UC_HOOK_CODE, self._hook_code, self)

        # Start emulation from the function start
        self.mu.emu_start(start_offset, 0)

        # Return value from RAX
        return self.mu.reg_read(UC_X86_REG_RAX)

    def _hook_external_call(self, name):
        print(f"[HOOK] External function: {name}")

        if name.startswith("memcpy"):
            dest = self.mu.reg_read(UC_X86_REG_RDI)
            src = self.mu.reg_read(UC_X86_REG_RSI)
            n = self.mu.reg_read(UC_X86_REG_RDX)
            self.mu.mem_write(dest, bytes(self.mu.mem_read(src, n)))

        elif name.startswith("strlen"):
            rdi = self.mu.reg_read(UC_X86_REG_RDI)
            rax = 0
            while self.mu.mem_read(rdi + rax, 1)[0] != 0:
                rax += 1
            self.mu.reg_write(UC_X86_REG_RAX, rax)

        else:
            print(f"[!] Unknown external call: {name}")
            self.mu.emu_stop()
            return

        # Simulate `ret` after external call (pop RIP)
        rsp = self.mu.reg_read(UC_X86_REG_RSP)
        ret_addr = int.from_bytes(self.mu.mem_read(rsp, 8), 'little')
        self.mu.reg_write(UC_X86_REG_RSP, rsp + 8)
        self.mu.reg_write(UC_X86_REG_RIP, ret_addr)
        print(f"Returning to 0x{ret_addr:X}")

    @staticmethod
```

```
    def _hook_code(uc, address, size, self):
        # Stop execution if return address is 0
        if address == 0:
            print("[*] Reached return address 0 — stopping emulation")
            self.mu.emu_stop()
            return

        # Call external hook if the address matches an import
        if address in self.hook_list:
            self._hook_external_call(self.hook_list[address])

    @staticmethod
    def get_imports():
        # Flatten all imported symbols into a list of (address, name)
        result = []
        for i in range(ida_nalt.get_import_module_qty()):
            def _cb(ea, name, ordinal):
                result.append((ea, name or f"ord_{ordinal}"))
                return True
            ida_nalt.enum_import_names(i, _cb)
        return result

    def get_string_at(self, addr):
        # Read a null-terminated string from memory
        i = 0
        while self.mu.mem_read(addr + i, 1)[0] != 0:
            i += 1
        return self.mu.mem_read(addr, i).decode()

    def dump(self, filename="/dev/shm/dump.bin"):
        with open(filename, "wb") as fd:
            fd.write(bytes(self.mu.mem_read(self.low_addr, self.length)))

my_runner = Runner()

# Run initialization function
my_runner.exec_func("init_phrases")

# Get the target address for the decrypt function
target = list(idautils.CodeRefsTo(idc.get_name_ea_simple("decrypt_phrase"), 0))[0]

# Feel free to dump to use with `strings`
my_runner.dump()

print("Decrypting strings...", hex(target))

for ref in idautils.CodeRefsTo(target, 0):
    print("Found ref:", hex(target))
    prev = ref
    offset = idc.BADADDR

    # Look backward up to 5 instructions to find `mov edi, imm`
    for _ in range(5):
        if (idc.print_insn_mnem(prev) == "mov" and
            idc.print_operand(prev, 0) == "edi" and
            idc.get_operand_type(prev, 1) == idc.o_imm):
            offset = idc.get_operand_value(prev, 1)
            break
        prev = idc.prev_head(prev)

    if offset == idc.BADADDR:
        print(f"[!] Could not find argument for call at {hex(ref)}")
        continue
```

```
    # Set EDI for decryption and run function
    my_runner.mu.reg_write(UC_X86_REG_EDI, offset)
    print(f"[+] Calling decrypt_phrase({offset}) at 0x{ref:X}")
    result_addr = my_runner.exec_func(target)

    # Read decrypted string
    decrypted = my_runner.get_string_at(result_addr)
    idc.set_cmt(ref, decrypted, 0)
    print(f"[*] Commented: '{decrypted}' at 0x{ref:X}")
```

About the author:

Pierre-Henri Pezier

Pierre-Henri Pezier is an IT Security Engineer and Threat Researcher with over a decade of experience in offensive security, reverse engineering, malware analysis and secure software development. He began reverse-engineering software in the early 2010s, a passion that expanded into analyzing advanced threats, developing decryptors, and writing detection rules. With a background in both offensive and defensive security, Pierre-Henri has worked on malware classification engines, sandbox environments, and EDR evasion techniques.
