Binary Hacking: X86 VM Sandbox Escape

Tizian Seehaus Athanasios Tsarapatsanis Murtaza Haidari
Beginner's Practical, B.Sc. Computer Science
Heidelberg University, Germany

***Abstract*—Our objective was to design an exploit within a memory management unit (MMU), which necessitated the development of our own emulated central processing unit (CPU).**

I. CPU Emulator - Architecture

A. Overview

In developing the CPU emulator (CPUE), we adhered to the principles and methodologies characteristic of Intel architecture. This approach led us to incorporate the following modules (Fig.[1]):

1) Embedded Arithmetic Logic Unit (ALU):
   Responsible for executing arithmetic and logical operations within the CPUE.
2) Memory Management Unit (MMU):
   Manages memory access and translation between virtual and physical addresses.
3) Translation Lookaside Buffer (TLB):
   Caches recent address translations to expedite memory access.
4) Memory-Mapped Input/Output (MMIO):
   Facilitates communication between the CPU and peripheral devices via designated memory addresses.
5) Programmable Interrupt Controller (PIC) & Interrupt Control Unit (ICU):
   Handles interrupt signals to ensure proper CPU response to asynchronous events.
6) Universal-Asynchronous-Receiver-Transmitter (UART) Controller:
   Manages serial communication by converting parallel data from the CPU into serial form and vice versa.
7) V100-Terminal:
   Emulates a terminal interface for user interaction with the CPUE.

B. Modules

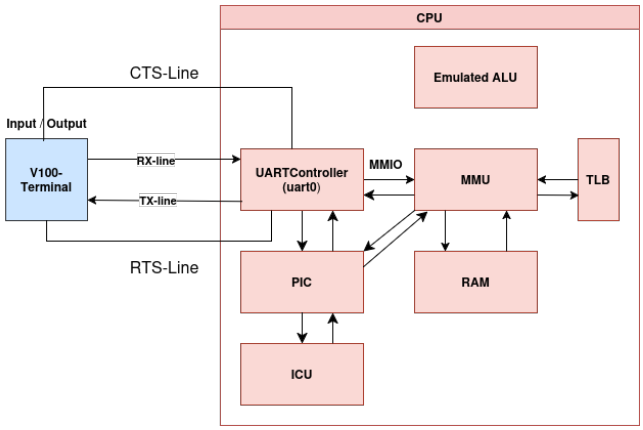

Fig. 1. Overview of CPU modules

1) Embedded ALU: This component represents the CPUE's internal logic capable of performing arithmetic and logical operations. Given the project's scope, implementing individual logic gates was deemed unnecessary. Instead, we implemented the official Intel instruction set architecture (ISA) [1] to define the ALU's functionality.

2) MMU: Our MMU implementation encompasses comprehensive memory management functionalities, excluding protected keys. It effectively handles page faults by delegating them to interrupt handlers, as detailed in the PIC section. The MMU incorporates an interrupt descriptor table and utilizes gate descriptors within the interrupt handler, supporting call, interrupt, and trap gates. It employs a four-level paging mechanism, aligning with Intel's architecture (Fig.[2]), and supports segmentation through logical addresses. Notably, the following code example show where our bug is hidden (List.[1]).

Listing 1. Exploit inside of the MMU

```
template<typename T = u8>
T* paddr_ptr(PhysicalAddress const& paddr) {
    CPUE_ASSERT(m_physmem != NULL, "m_physmem==NULL");
    return (T*)(m_physmem + paddr.addr);
}
```

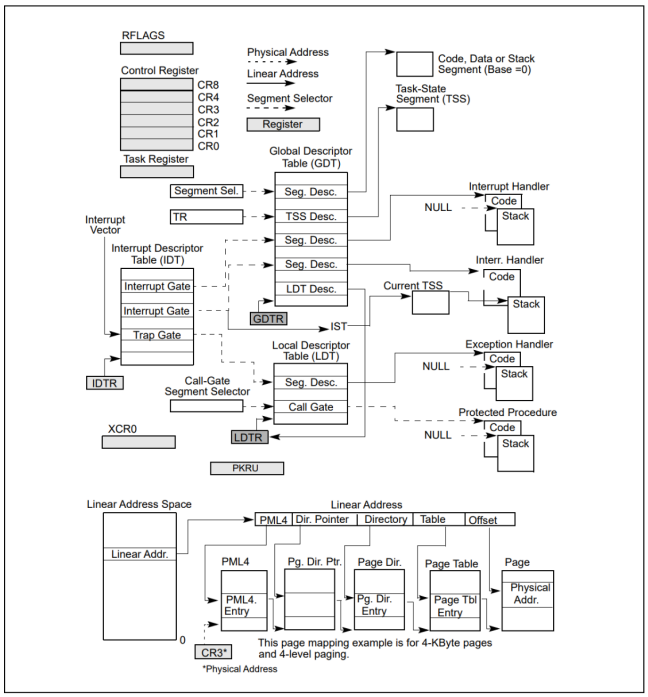

Figure 2-2. System-Level Registers and Data Structures in IA-32e Mode and 4-Level Paging

Fig. 2. General paging structure

3) TLB: The TLB is designed following Intel's standards to enhance MMU performance by caching recent address translations, thereby reducing memory access latency.

4) MMIO: This module maps registers inside of the PIC and the UART-controller to the MMU, facilitating communication between hardware components and the CPU. This ensures for a data exchange between the MMU and the PIC, as well as between the MMU and the UART-controller. (Fig. [3])

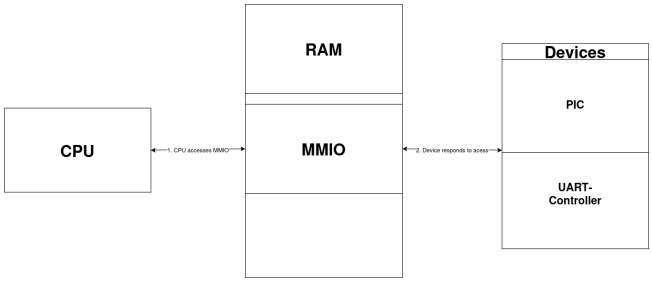

Fig. 3. Depiction of MMIO

5) PIC & ICU: The design of our PIC, was inspired by the 8259A PIC. Our system features extensive interrupt handling capabilities, allowing for nested interrupts that are processed according to their priority levels. When an interrupt occurs during the handling of another, the system manages them based on predefined priorities. Additionally, we support hardware interrupt logic, which necessitates management by the operating system's kernel [2].

6) UART-Controller: The design pattern was inspired by the TL16C750 model. It facilitates serial communication through Receive/Transmit lines, along with clear-to-send and ready-to-send lines. Regarding the original design ours only supports the FIFO-mode [3].

7) V100-Terminal: The V100-Terminal emulates a simple terminal interface, enabling user interaction by providing input and output functionalities. It also provides an internal FIFO-buffer which caches in characters that are not ready to be received by the UART-controller.

C. Executing the CPU-Emulator

When operating the CPUE with a mini-kernel, we successfully executed various programs. This includes running tiny BASIC programs, such as a simple bubble-sort algorithm, as well as applications like 'cowsay' and others that require minimal system calls (Fig. [4] & [5]).

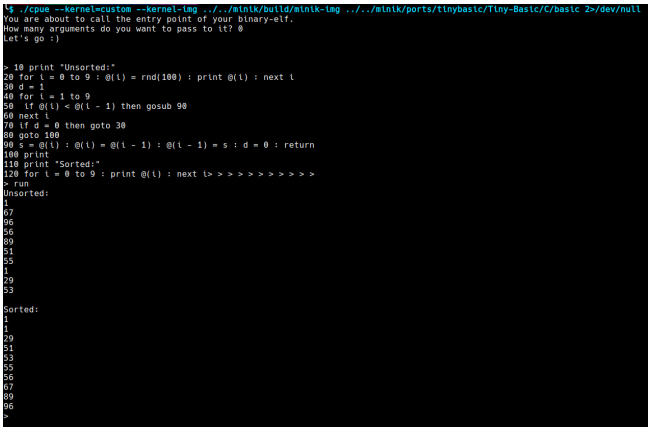

Fig. 4. Executing bubble-sort on tiny-basic

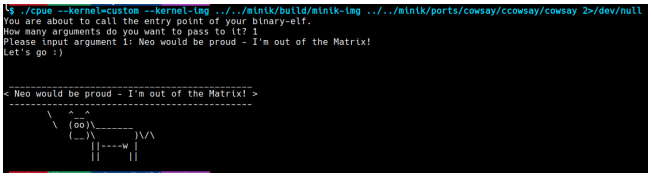

Fig. 5. Executing cowsay

II. Exploiting the emulator

A. The idea behind the exploit

Our exploit draws inspiration from the return-to-libc attack technique. Specifically, we aim to manipulate the exit_func_list entries to include a call to system("/bin/sh"). This manipulation is facilitated by the given vulnerability within our Memory Management Unit (MMU) implementation, which fails to enforce proper bounds checking on physical addresses. Consequently, this oversight permits an overflow of the emulated RAM, allowing access to the standard C library (libc) within the process's memory space. Notably, our approach does not involve disabling existing security mechanisms such as stack canaries, Address Space Layout Randomization (ASLR), etc. (Fig. [6])

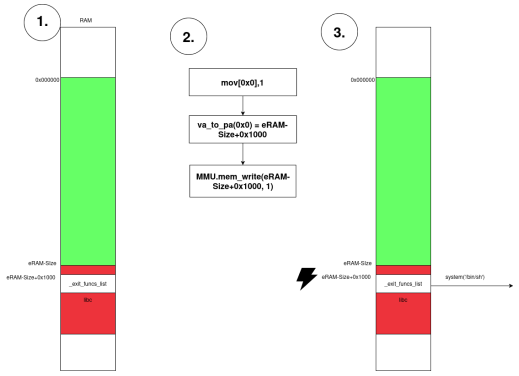

Fig. 6. Overview of the stack

B. How to run the exploit

1) The Decryption of mangled pointer: In order to execute the exploit, we must first locate the relative offset of the libc-library to our emulated RAM. This step is crucial because ASLR is randomizing the address each time we run the program. To achieve this we randomly jump out of the emulated RAM's bounds and try calculate the relative offset. When inside of libc we can now read the entries of the global offset table (GOT) and bypass ASLR.

After finding the libc base we can now calculate the location of `__exit_function_list` and thus search for the entry `_dl_fini`. We can detect `_dl_fini` as it's values are `flavor=ef_cxa, arg=0, dso_handle=0`. By finding `_dl_fini`, which is the secret for mangling the fn-pointer, we are now able to decrypt the fn-pointer. Decrypting the fn-pointer is possible by reversing the encryption-steps (Encryption of fn-pointer). So in our case we have to do a rotation by 0x11 of the `fn-pointer` and then XOR it with `dl_fini_addr`.

2) Overwriting exit_function: Now we can overwrite the `exit_function.fn` member of the `exit_function` struct entry corresponding to `_dl_fini` with our own correctly mangled system function pointer. As an argument we set `exit_function.arg` to the address of the string `"/bin/sh"`, which is conveniently already present inside the libc data segment. This results in an overall effect wherein the libc exit handler routines will handle our faked entry and call `system("/bin/sh")`, giving us remote code execution on the hypervisor host machine, and with this completing our x86 vm sandbox escape.

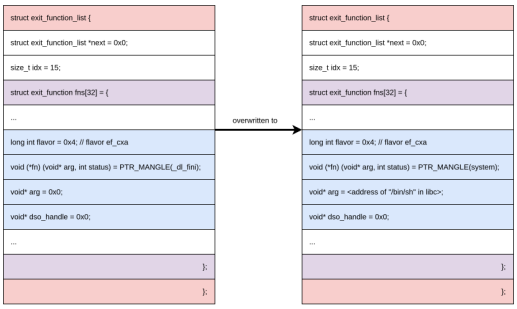

Fig. 7. Overview of exit_func_list

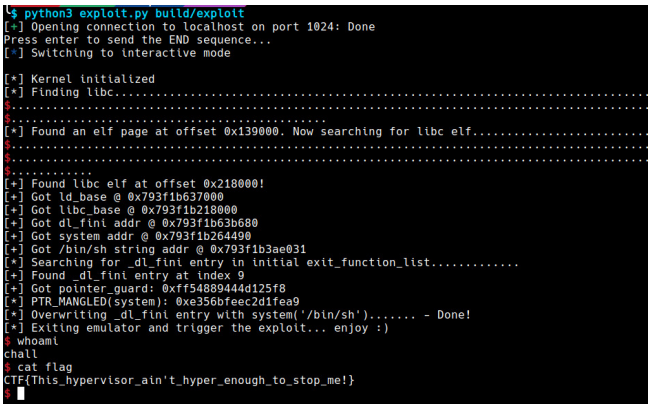

Fig. 8. Running the Exploit

REFERENCES

[1] *Intel® 64 and IA-32 Architectures Software Developer's Manual Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4*, Intel Corporation, December 2024, available at: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html.

[2] *Intel® 8259A Programmable interrupt controller (8259A/8259A-2)*, Intel Corporation, available at: https://pdos.csail.mit.edu/6.828/2010/readings/hardware/8259A.pdf.

[3] *PC16550D Universal Asynchronous Receiver/Transmitter With FIFOs*, Texas Instruments Incorporated, available at: https://media.digikey.com/pdf/Data%20Sheets/Texas%20Instruments%20PDFs/PC16550D.pdf.