Payment terminals as general purpose (game-)computers

Thomas Rinsma

MCH2022

2022-07-25

Table of contents

Disclaimers

Quick heads up

- No new vulnerabilities in these slides
- We will ignore payment keys / payment security

whoami

Thomas Rinsma
Security Analyst @ Codean

Background:

- Computer science, software security
- Android, mobile payments
- Getting into embedded

`https://th0mas.nl`
`thomasrinsma@protonmail.com`

Context

I was bored and went looking for a target:

- Embedded system to run Doom on
- Not too crazy in terms of hardware hacking skills
- Cool factor?

The device

VX820

The device

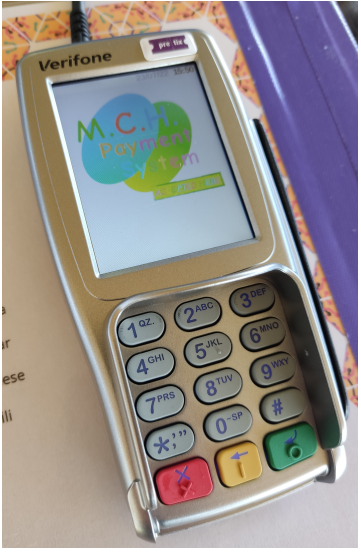
Why?

Why a payment terminal?

- Seems unnecessarily powerful
- All the useful peripherals

Why this device?

- Relatively old: easier exploitation?
- Still quite common in NL



Device specs

Hardware:

- 400Mhz ARMv6 processor
- 128MB flash, 32MB RAM
- 240x320 color LCD (touchscreen)!
- Ethernet, USB, serial
- Smartcard reader (x4), NFC, magstripe, beeper

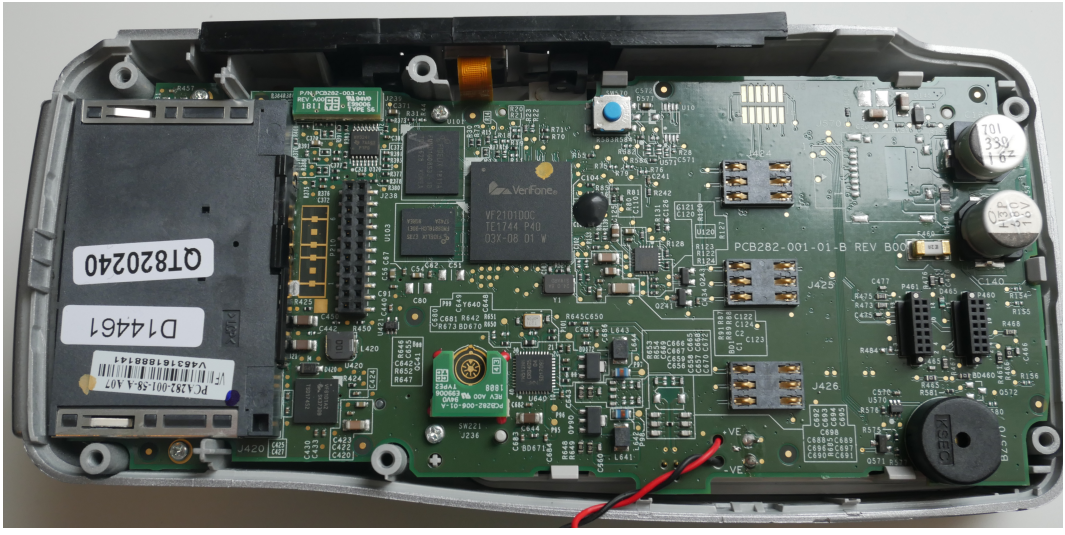
Software:

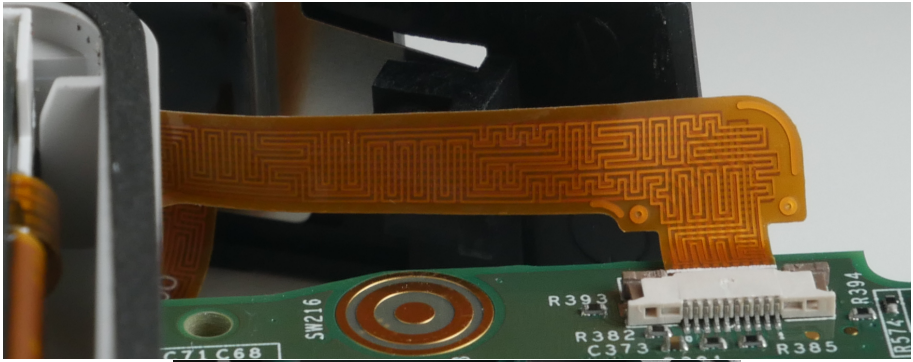
- "Verix OS"
- Multi-application
- Configuration through env vars
- Unix-like filesystem/syscalls

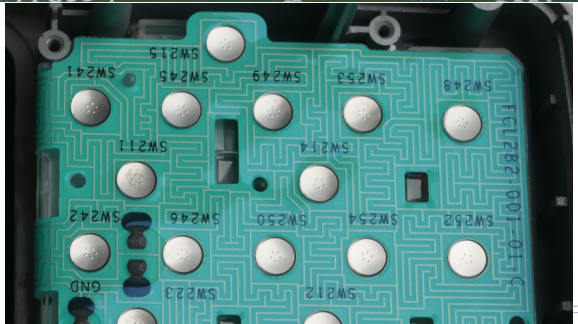

Verifone
San Jose, CA USA
Model:VX820
Rating:9VDC, 1A
Product:VX820 CTLS
R-NZ
Prod No.:M282-704-HB-EUA-3 REV.B11
MADE IN MALAYSIA VEN: AE17
S/N: 903-117-157
PTID: 76132133
10BT/HS: 60C798FBA751
ITS
9-8-2018
Sw.V: 11.3.06
Sec.V: 11.07
ITS
EIJSINK
Mach.nr
216645

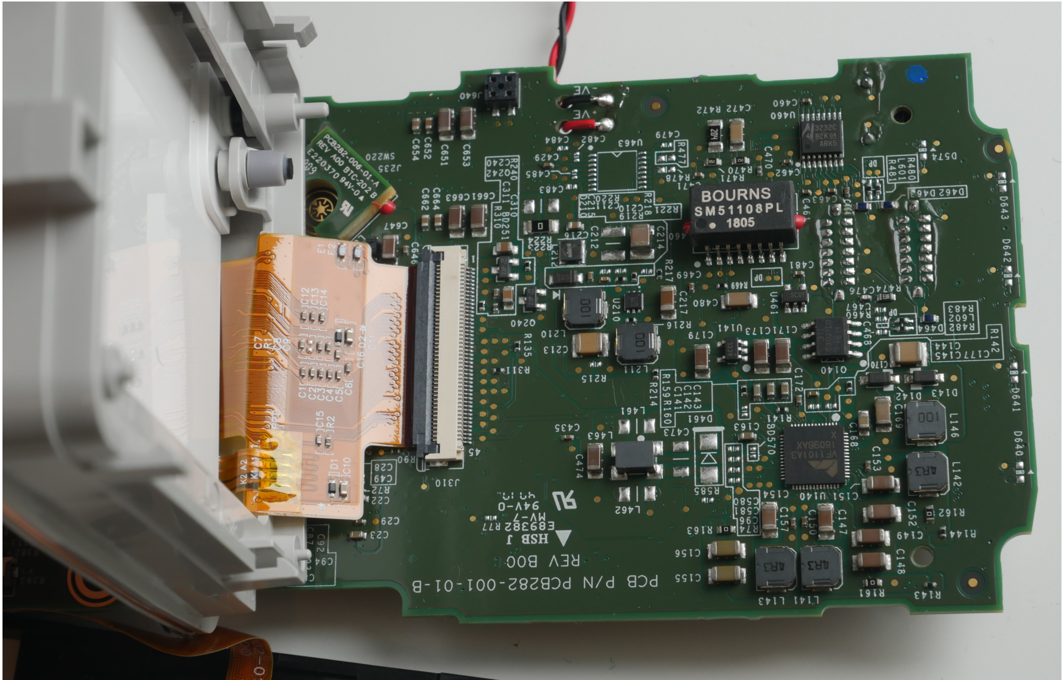

1
2
3
Verifone
San Jose, CA USA
Model:VX820
Rating:9VDC, 1A
Product:VX820 CTLS
R-NZ
Prod No.:M282-704-HB-EUA-3 REV.B11


J240
0203.18.
PCB282-001-01-B REV B0C
J424
J425
J426
J570
J3
C160
GPSK


VeriFone
VF210100C
TE1744 P40
03X-08 01 W
PCB282-001-01-B REV B00
J425
J426
J420
+VE
-VE
Q18202 40
D14461


SW216
R393
R394
R382
C373
R385
C71 C68


SW241
SW215
SW245
SW249
SW253
SW248
SW211
SW254
SW242
SW246
SW250
SW254
SW252
GND
SW223
SW212


BOURNS
SM51108PL
1805
PCB P/N PCB282-001-01-B
REV B00

Getting access

How it all began

Getting access

Initial state:

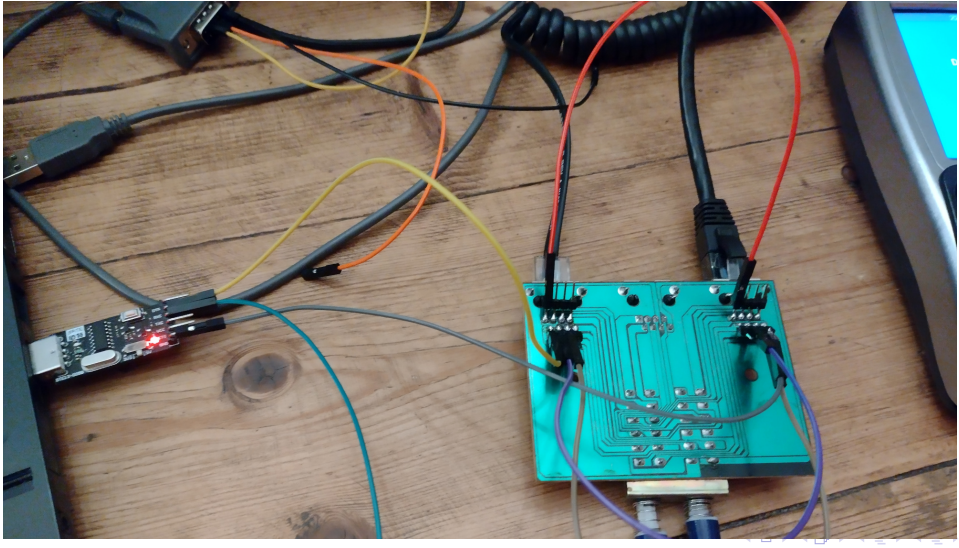
- Running the CCV payment application
- Configured as a “pin pad”

Locked down..

- No way to exit application
- System menu shortcut disabled
- Only processing commands from POS

Getting access

Capturing the commands between the VX570 and the VX820



Getting access

Some time later :)

Getting access

But this doesn't get us any closer to Doom...

Getting access

These devices wipe sensitive data when tampered with.

So what happens when we open it up?

Getting access

A different screen!

- Device is fully wiped
- Now boots into system menu

System menu password is publicly documented :)

Getting access

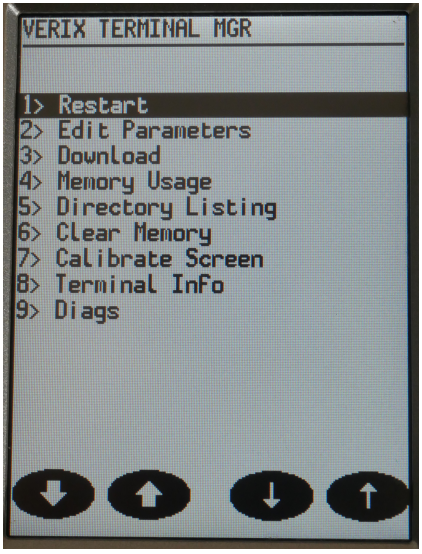
A different screen!

- Device is fully wiped
- Now boots into system menu

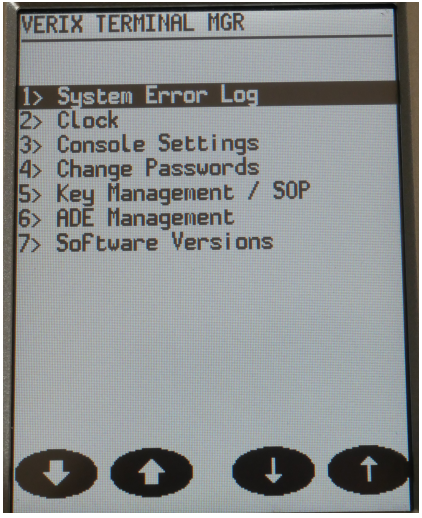
System menu password is publicly documented :)

166831

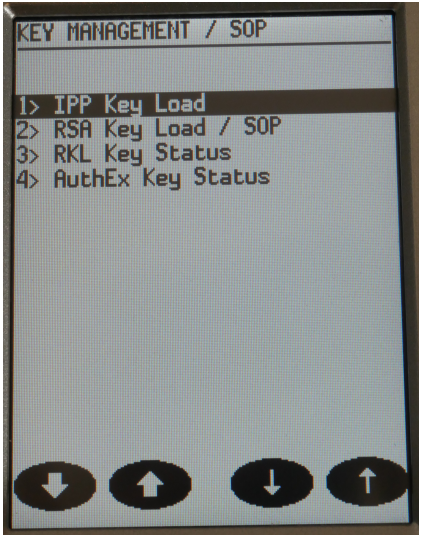
Clearing the tamper flag



Clearing the tamper flag



Clearing the tamper flag



Clearing the tamper flag

Clearing the tamper flag

- Press '2' on `IPP Key Load` screen

Clearing the tamper flag

- Press '2' on `IPP Key Load` screen
- Device reboots

Clearing the tamper flag

- Press '2' on `IPP Key Load` screen
- Device reboots
- No longer "tampered"!

Getting access

Downloading applications

`DOWNLOAD NEEDED?`

- Proprietary protocol called XDL
- Features:
  - Load files
  - Set config variables
  - Wipe flash or SRAM

Getting access

Downloading applications

DOWNLOAD NEEDED?

- Proprietary protocol called XDL
- Features:
  - Load files
  - Set config variables
  - Wipe flash or SRAM

Easily reverse engineered :)

```
1  from xdl import XDL
2
3  binary = "APP.OUT"
4
5  xdl = XDL()
6
7  xdl.connect()
8  xdl.set_config_var("*GO", binary)
9  xdl.send_file(binary)
10 xdl.stop()
```

https://github.com/ThomasRinsma/pyxdl

Getting access

So, can we just upload `DOOM.OUT`?

Getting access

Program authentication

Yes, but...

- Programs (`.OUT`) normally come with a signature file (`.P7S`)
- Replaced with a `.S1G` file after first boot.
- On boot:

```
if(verify_p7s(file))
   generate_s1g(file);
```

- Runtime:

```
verify_s1g(file);
```

Source: research by `@ivachou` and `@A1ex_S`:
`https://www.paymentvillage.org/resources`

Known issues

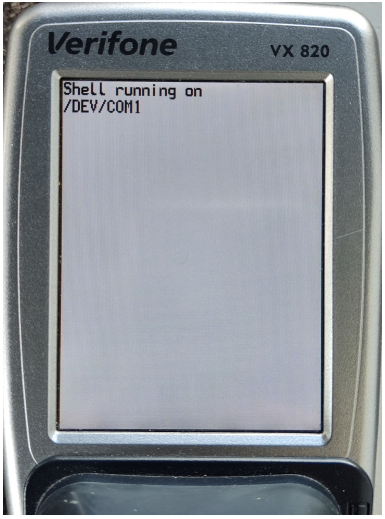
Previously found "features"/bugs:

Source: research by `@ivachou` and `@A1ex_S`:
`https://www.paymentvillage.org/resources`

Known issues

Previously found "features"/bugs:

1. Hidden shell: `T:SHELL.OUT`
   - interesting but low privilege



Source: research by `@ivachou` and `@A1ex_S`:
`https://www.paymentvillage.org/resources`

Known issues

Previously found "features"/bugs:

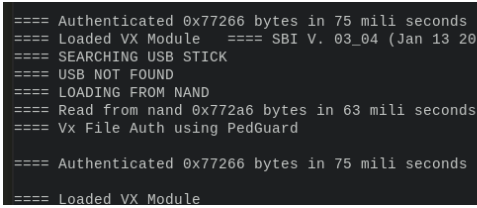
1. Hidden shell: `T:SHELL.OUT`
   - interesting but low privilege
2. Buffer overflow in kernel code
   - patched or different per OS version

Source: research by `@ivachou` and `@A1ex_S`:
`https://www.paymentvillage.org/resources`

Known issues

Previously found "features"/bugs:

1. Hidden shell: `T:SHELL.OUT`
   - interesting but low privilege
2. Buffer overflow in kernel code
   - patched or different per OS version
3. Hidden bootloader mode
   - **Still present in this device!**



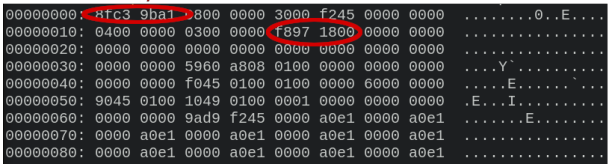
Source: research by `@ivachou` and `@A1ex_S`:
`https://www.paymentvillage.org/resources`

Boot sequence

Overview

- "Secure boot": each stage authenticates the next
- 2nd stage (SBI) authenticates and loads Verix OS
- SBI also listens for a keycombo: **1**+**5**+**9**
  - Uses XDL to load authenticated *scripts*
  - Or, if a magic header is provided:

magic addr



`memcpy(addr, file_contents, file_len)`

Boot sequence

Summary

To summarize:

- Arbitrary write allows for code execution
- Completely breaking secure boot

Boot sequence

Summary

To summarize:

- Arbitrary write allows for code execution
- Completely breaking secure boot
- Somehow still present on these devices!

Boot sequence

Summary

To summarize:

- Arbitrary write allows for code execution
- Completely breaking secure boot
- Somehow still present on these devices!

- Luckily for us: this is a way in :)

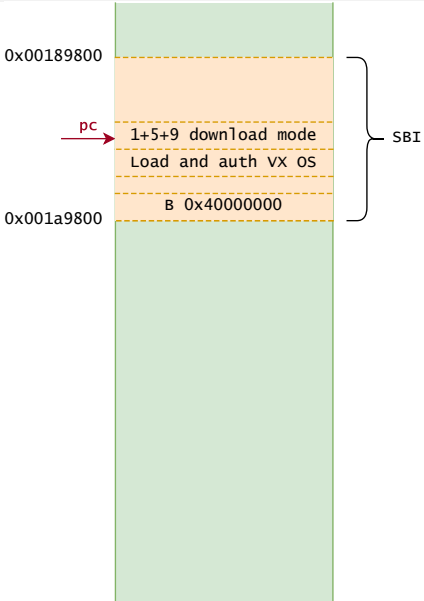
Code execution

The plan

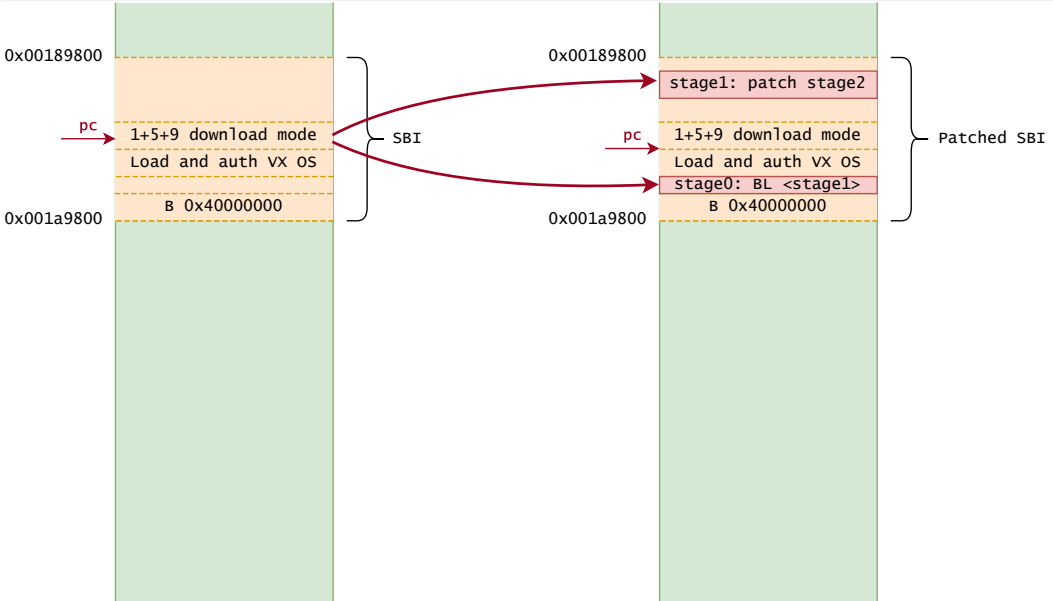
Use code execution in SBI to get control over Verix:

1. Overwrite SBI with a patched version
2. Keep original bootloader functionality intact
3. Add a patch to the OS that calls `gen_s1g("H.OUT");`

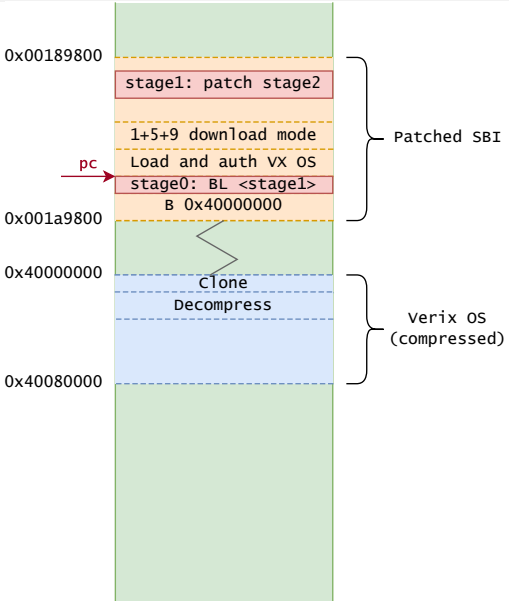
Code execution (1)



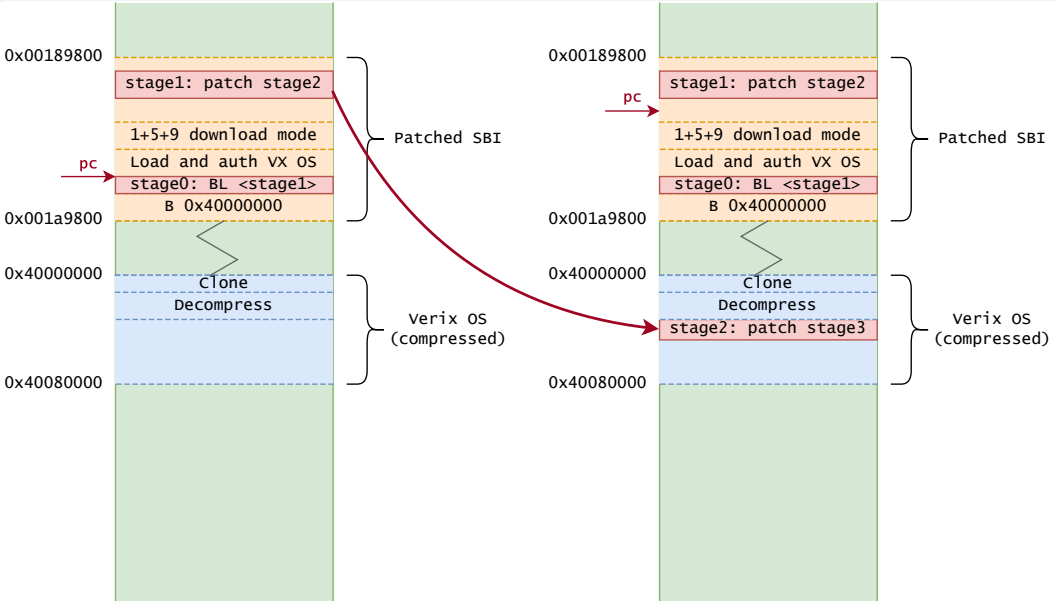
Code execution (1)



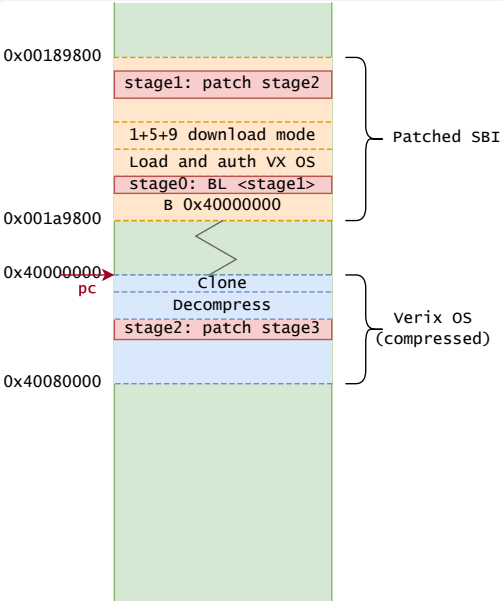
Code execution (2)



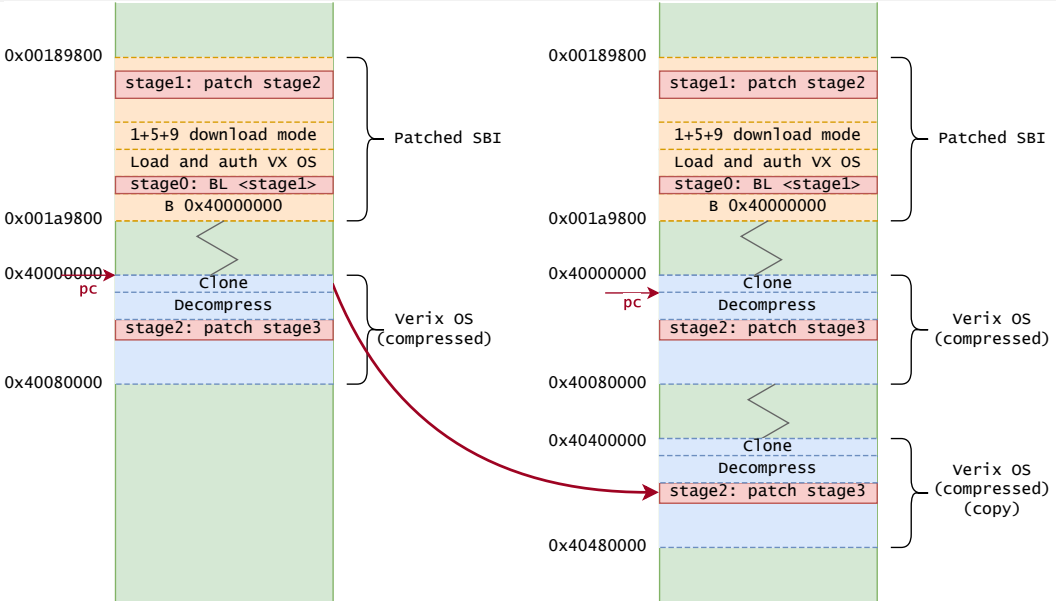
Code execution (2)



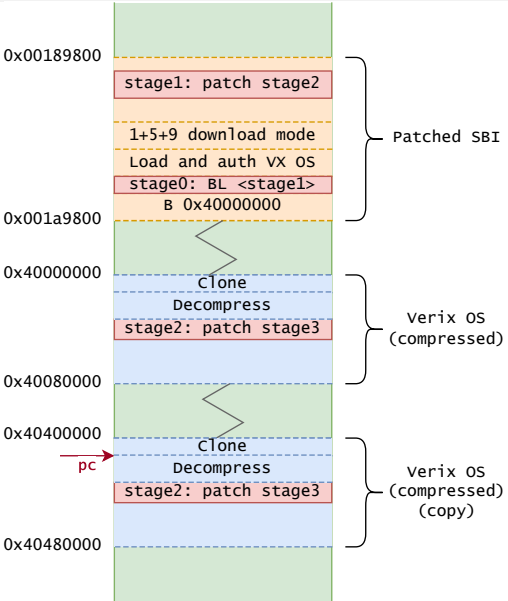
Code execution (3)



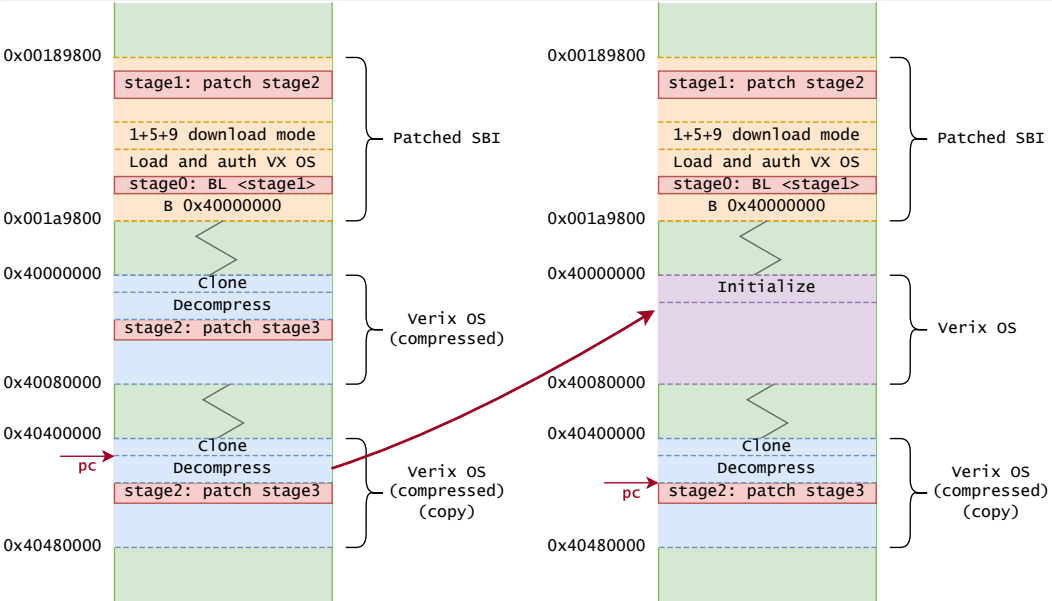
Code execution (3)



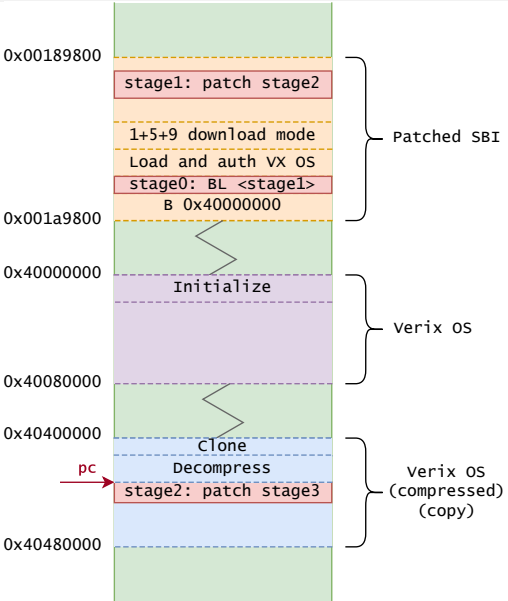
Code execution (4)



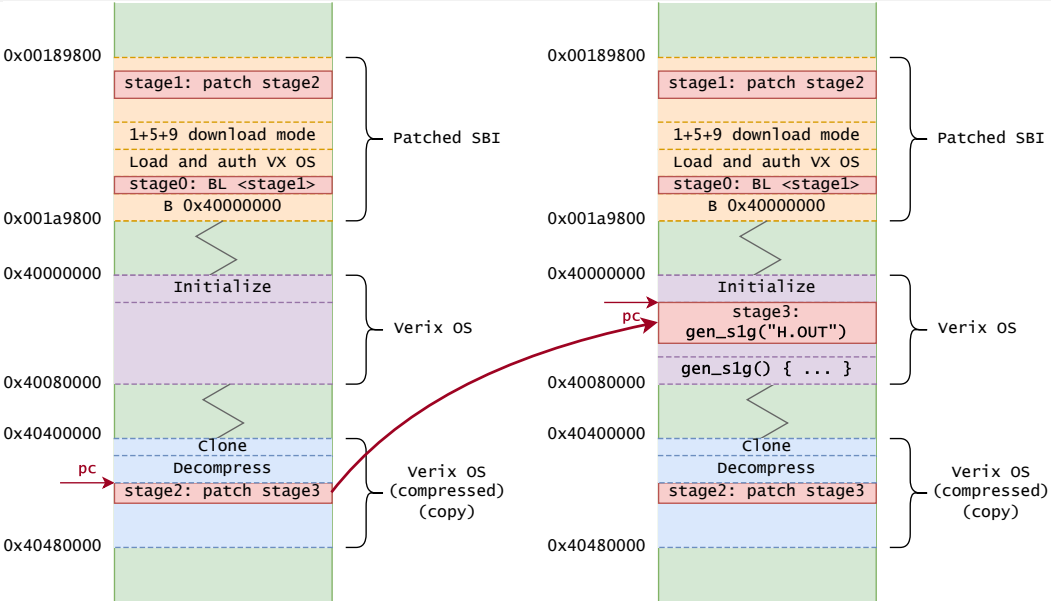
Code execution (4)



Code execution (5)

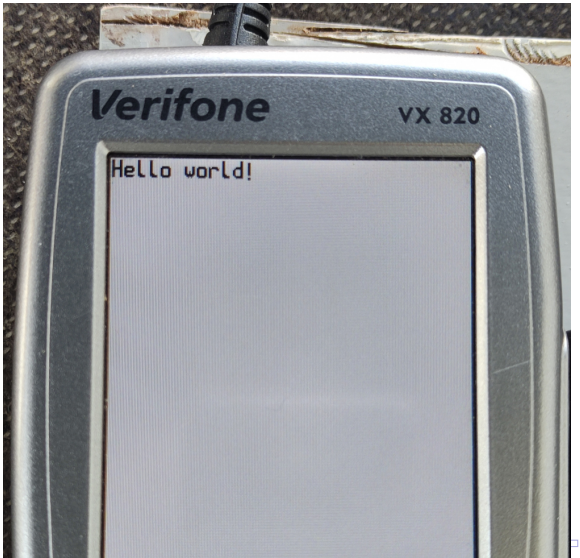


Code execution (5)



Code execution

The result



Executable format

```
00000000: a307 0100 0000 aabb 7042 0040 0080 0000  ........pB.@....
00000010: 0000 8000 0000 0000 0080 0000 7042 0075  ............pB.u
00000020: 2e53 5953 2e4c 4942 0000 0000 0000 0000  .SYS.LIB........
00000030: 0000 0000                                ....
```

The program header specifies:

- Magic and various flags (?)
- Entrypoint
- Which system libraries to load (e.g. `SYS.LIB`)
- Start and size of code (ELF's `.text`)
- Size of read-only data (ELF's `.rodata`)
- Stack size

Toolchain

Format seems pretty simple!

Toolchain

Format seems pretty simple!

Let's make a hacky "toolchain":

- Build an ARMv6 ELF file normally
- Extract the relevant sections
- Copy-paste the header and patch the sizes

```
ENTRY(main)
OUTPUT_FORMAT(elf32-bigarm)

SECTIONS
{
  . = 0x70420074;
  .text : { *(.text) }
  .data : { *(.data) }
  .bss : { *(.bss) }
}
```

```
$(OUT_FILE): $(OBJECTS)
    # link
    $(LD) $(OBJECTS) -T $(TOOLS_DIR)/script.ld -o $@_elf

    # copy just .text, .rodata and .bss
    $(OBJCOPY) -O binary $@_elf $@_binary

    # add header and create OUT file
    cat $(TOOLS_DIR)/base.out $@_binary > $@

    # fix header fields
    python $(TOOLS_DIR)/fix_size.py $(OUT_FILE)
```

Toolchain

Syscalls

We want to be able to print to the screen, read input, etc. $\longrightarrow$ **syscalls**.

- The interface is familiar: `open`, `read`, `write`, etc.
  - print to screen: `write` to `/DEV/CONSOLE`
  - read keystrokes: `read` from `/DEV/CONSOLE`
- Syscall numbers can be RE'd from other programs and public documentation

Toolchain

Now it's a matter of engineering:

```
int open(const char *path, int flags) {
    int ret;

    asm volatile(
        "mov r0, %[path]\n"
        "mov r1, %[flags]\n"
        "svc 5\n"
        "mov %[ret], r0\n"
        :
            [ret] "=l" (ret)
        :
            [path] "l" (path),
            [flags] "l" (flags)
        :
            "r0", "r1", "memory"
    );

    return ret;
}
```

```
int read(int fd, char *buf, unsigned len) {
    int ret;

    asm volatile(
        "mov r0, %[fd]\n"
        "mov r1, %[buf]\n"
        "mov r2, %[len]\n"
        "svc 1\n"
        "mov %[ret], r0\n"
        :
            [ret] "=l" (ret)
        :
            [fd] "l" (fd),
            [buf] "l" (buf),
            [len] "l" (len)
        :
            "r0", "r1", "r2", "memory"
    );

    return ret;
}
```

```
int write(int fd, const char *buf, unsigned len) {
    int ret;

    asm volatile(
        "mov r0, %[fd]\n"
        "mov r1, %[buf]\n"
        "mov r2, %[len]\n"
        "svc 0\n"
        "mov %[ret], r0\n"
        :
            [ret] "=l" (ret)
        :
            [fd] "l" (fd),
            [buf] "l" (buf),
            [len] "l" (len)
        :
            "r0", "r1", "r2", "memory"
    );

    return ret;
}
```

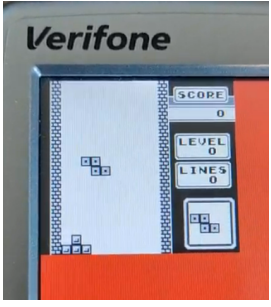
etcetera…

Porting stuff

Now we can start porting Doom :)

[demo time]

The end



@thomasrinsma

https://th0mas.nl/2022/07/18/verifone-pos-hacking/