

Payment terminals as general purpose (game-)computers

Thomas Rinsma

MCH2022

2022-07-25

Table of contents

- ▶ Disclaimers + introduction
- ▶ About the device
- ▶ Getting access
- ▶ Bootloader and OS
- ▶ Building a "toolchain"
- ▶ Porting Doom and more
- ▶ Demo time

Disclaimers

Quick heads up

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

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

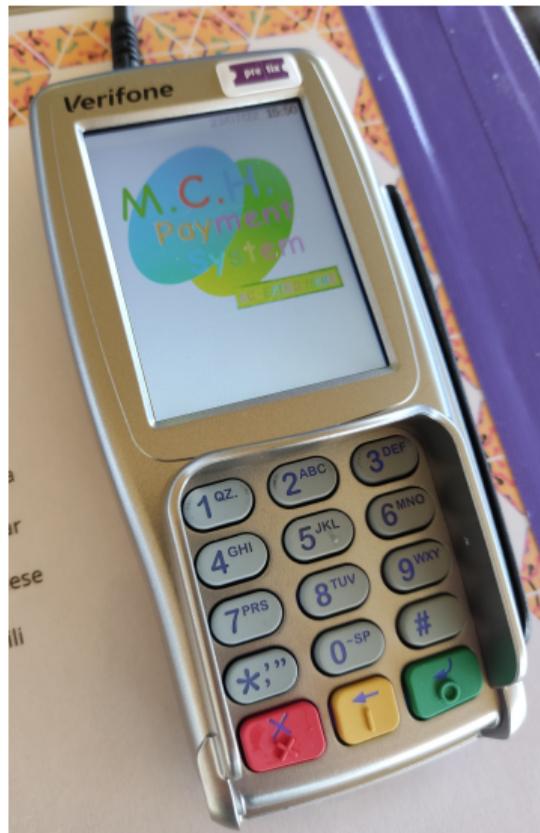
Why?

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

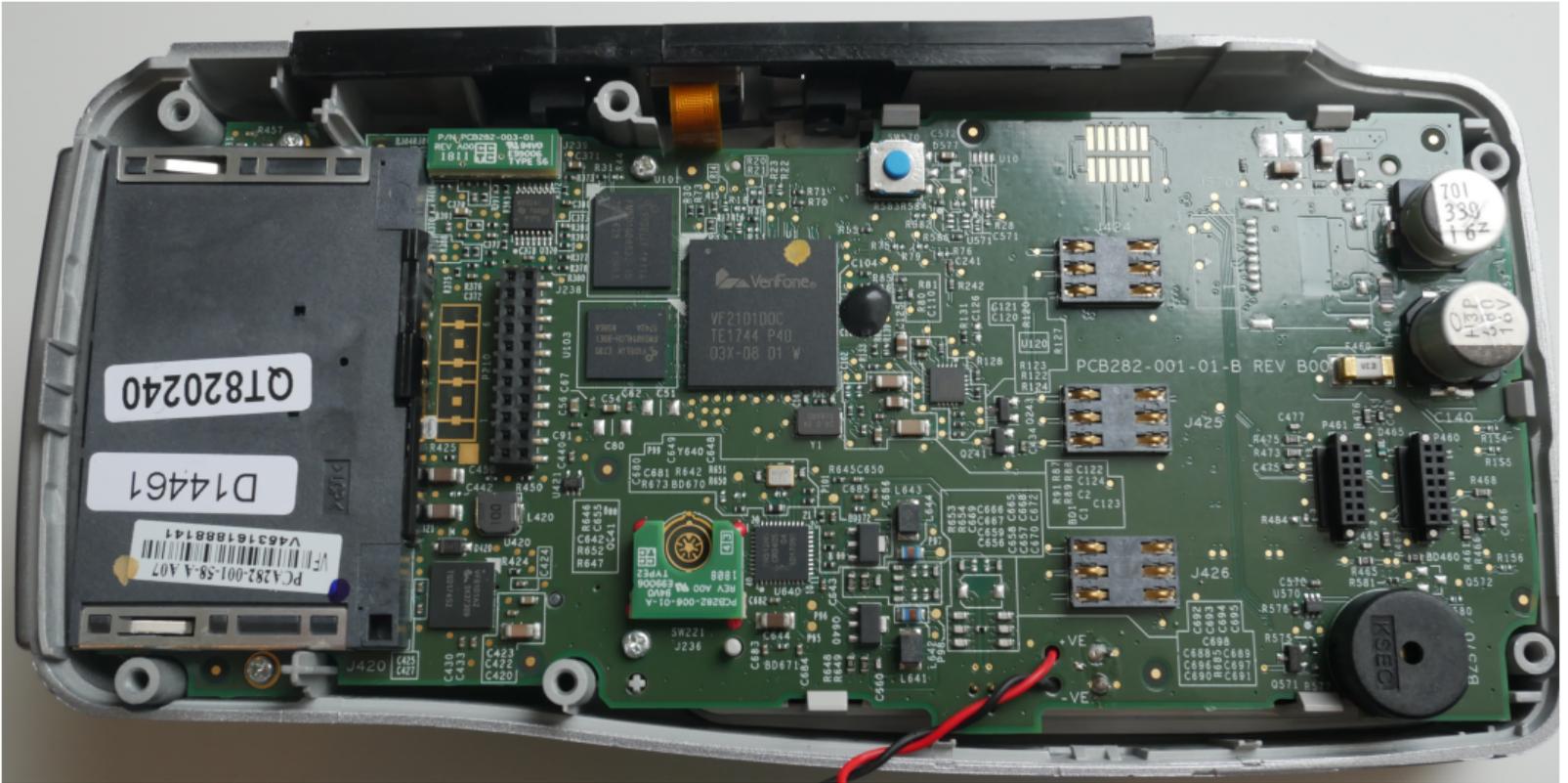
Software:

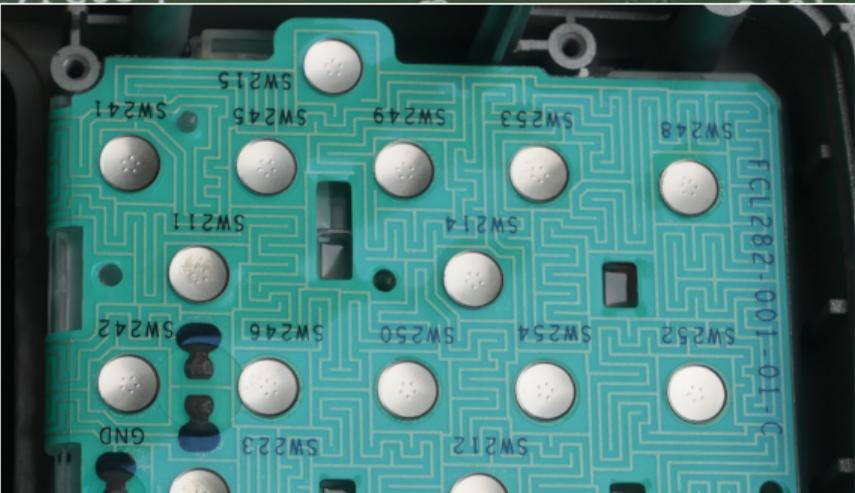
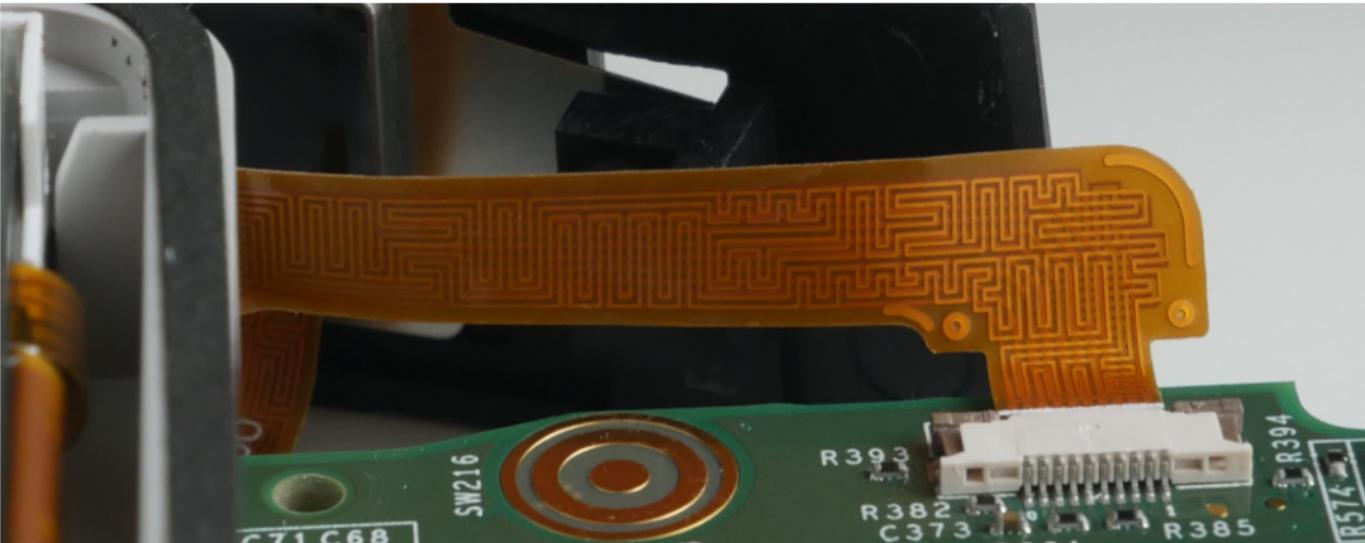
- ▶ "Verix OS"
- ▶ Multi-application
- ▶ Configuration through env vars
- ▶ Unix-like filesystem/syscalls



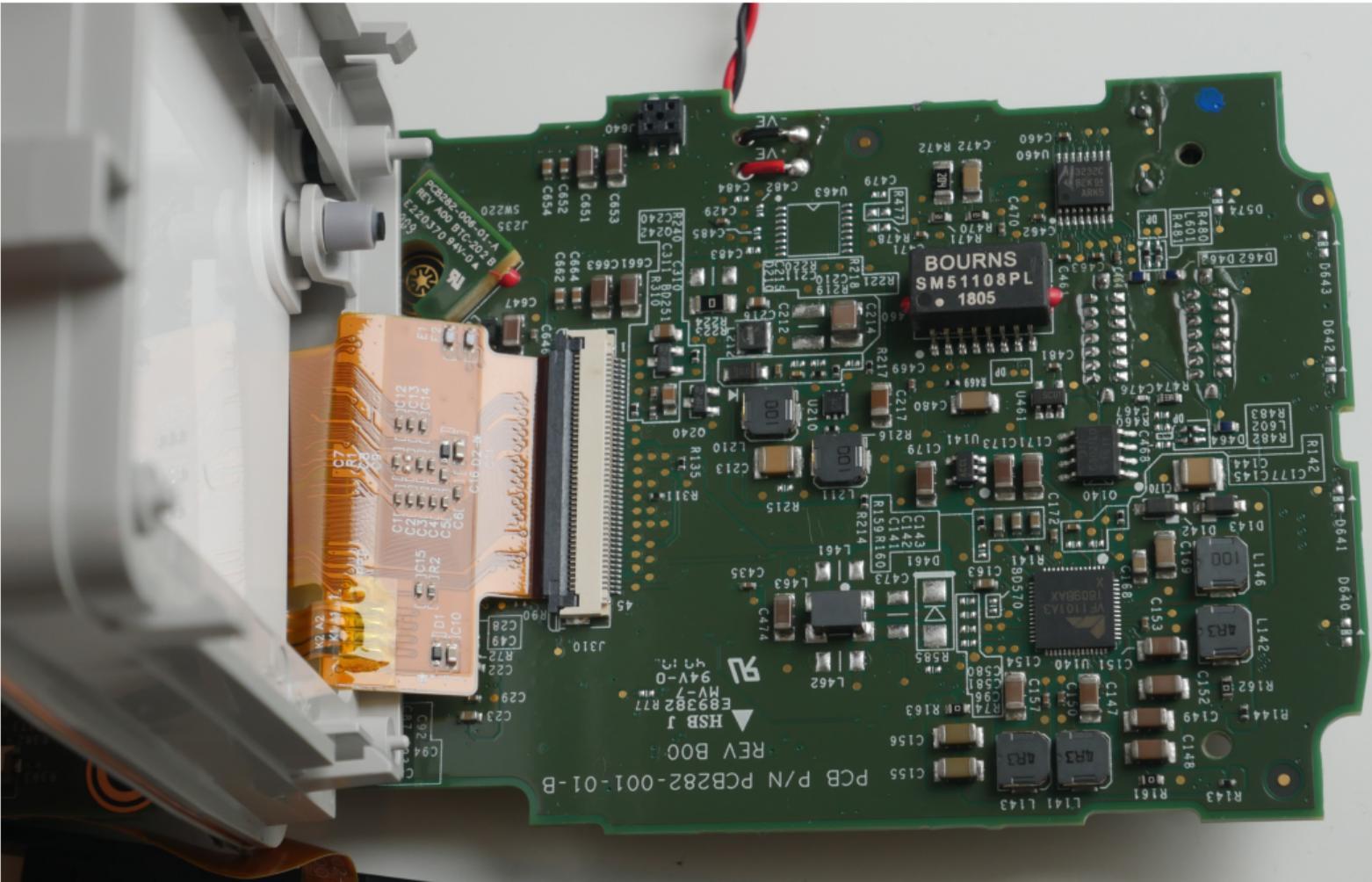












PCB P/N PCB282-001-01-B

REV B00

HSB J

E89382 R7

MV-0

94V-0

UL

51 13

G9 A C

G2 A C

C29

C22

R72

C49

R2

C15

D1

C10

C6

C16 D2-3

C1

IC12

IC13

IC14

IC15

IC16

IC17

IC18

IC19

IC20

IC21

IC22

IC23

IC24

IC25

IC26

IC27

IC28

IC29

IC30

IC31

IC32

IC33

IC34

IC35

IC36

IC37

IC38

IC39

IC40

IC41

IC42

IC43

IC44

IC45

IC46

IC47

IC48

IC49

IC50

IC51

IC52

IC53

IC54

IC55

IC56

IC57

IC58

IC59

IC60

IC61

IC62

IC63

IC64

IC65

IC66

IC67

IC68

IC69

IC70

IC71

IC72

IC73

IC74

IC75

IC76

IC77

IC78

IC79

IC80

IC81

IC82

IC83

IC84

IC85

IC86

IC87

IC88

IC89

IC90

IC91

IC92

IC93

IC94

IC95

IC96

IC97

IC98

IC99

IC100

IC101

IC102

IC103

IC104

IC105

IC106

IC107

IC108

IC109

IC110

IC111

IC112

IC113

IC114

IC115

IC116

IC117

IC118

IC119

IC120

IC121

IC122

IC123

IC124

IC125

IC126

IC127

IC128

IC129

IC130

IC131

IC132

IC133

IC134

IC135

IC136

IC137

IC138

IC139

IC140

IC141

IC142

IC143

IC144

IC145

IC146

IC147

IC148

IC149

IC150

IC151

IC152

IC153

IC154

IC155

IC156

IC157

IC158

IC159

IC160

IC161

IC162

IC163

IC164

IC165

IC166

IC167

IC168

IC169

IC170

IC171

IC172

IC173

IC174

IC175

IC176

IC177

IC178

IC179

IC180

IC181

IC182

IC183

IC184

IC185

IC186

IC187

IC188

IC189

IC190

IC191

IC192

IC193

IC194

IC195

IC196

IC197

IC198

IC199

IC200

IC201

IC202

IC203

IC204

IC205

IC206

IC207

IC208

IC209

IC210

IC211

IC212

IC213

IC214

IC215

IC216

IC217

IC218

IC219

IC220

IC221

IC222

IC223

IC224

IC225

IC226

IC227

IC228

IC229

IC230

IC231

IC232

IC233

IC234

IC235

IC236

IC237

IC238

IC239

IC240

IC241

IC242

IC243

IC244

IC245

IC246

IC247

IC248

IC249

IC250

IC251

IC252

IC253

IC254

IC255

IC256

IC257

IC258

IC259

IC260

IC261

IC262

IC263

IC264

IC265

IC266

IC267

IC268

IC269

IC270

IC271

IC272

IC273

IC274

IC275

IC276

IC277

IC278

IC279

IC280

IC281

IC282

IC283

IC284

IC285

IC286

IC287

IC288

IC289

IC290

IC291

IC292

IC293

IC294

IC295

IC296

IC297

IC298

IC299

IC300

IC301

IC302

IC303

IC304

IC305

IC306

IC307

IC308

IC309

IC310

IC311

IC312

IC313

IC314

IC315

IC316

IC317

IC318

IC319

IC320

IC321

IC322

IC323

IC324

IC325

IC326

IC327

IC328

IC329

IC330

IC331

IC332

IC333

IC334

IC335

IC336

IC337

IC338

IC339

IC340

IC341

IC342

IC343

IC344

IC345

IC346

IC347

Getting access

How it all began



Getting access

Initial state:

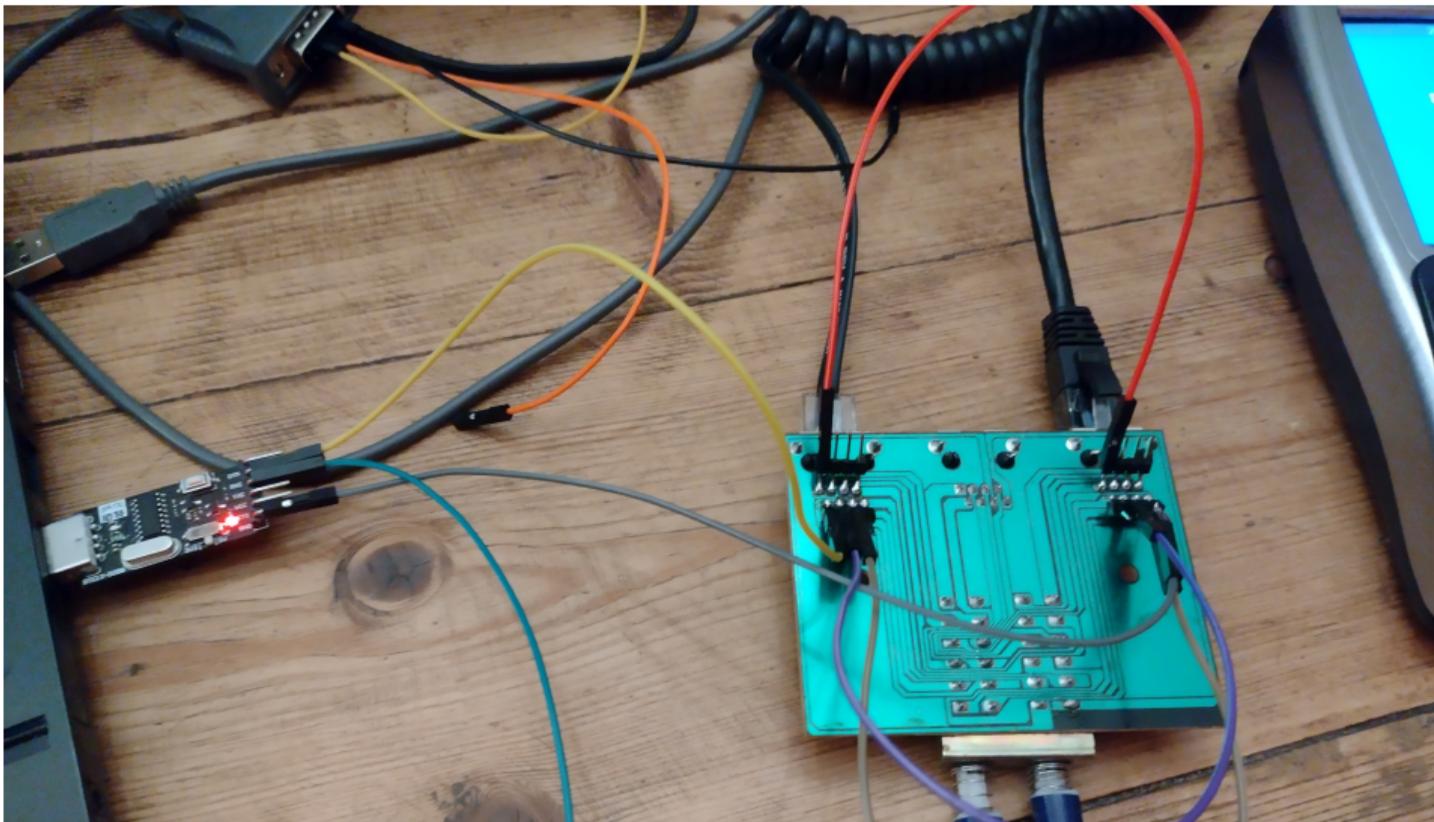
- ▶ Running the CCV payment application
- ▶ Configured as a “pin pad”

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!

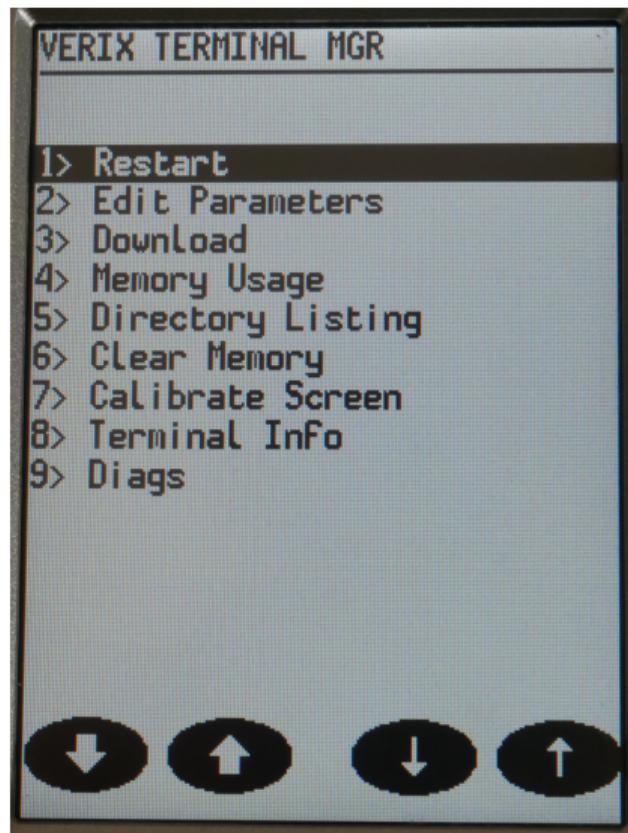
- ▶ Device is fully wiped
- ▶ Now boots into system menu

System menu password is publicly documented :)

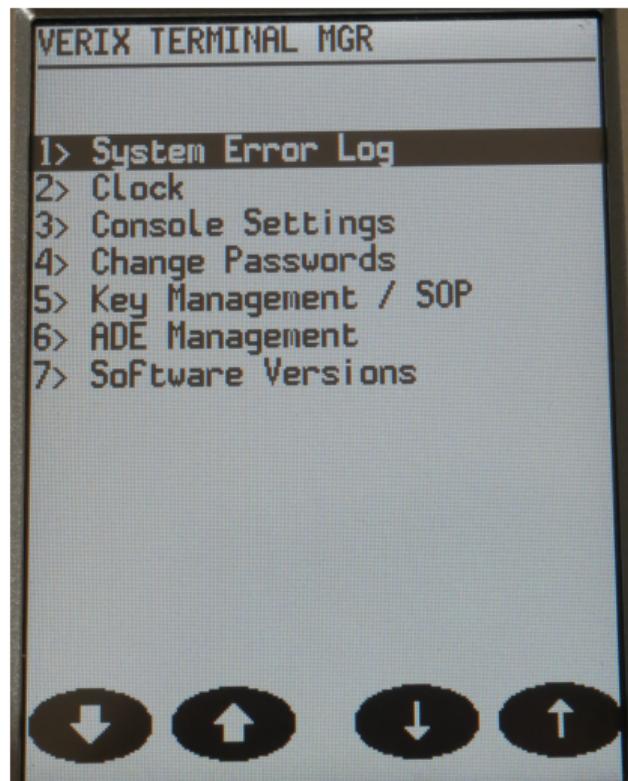
166831



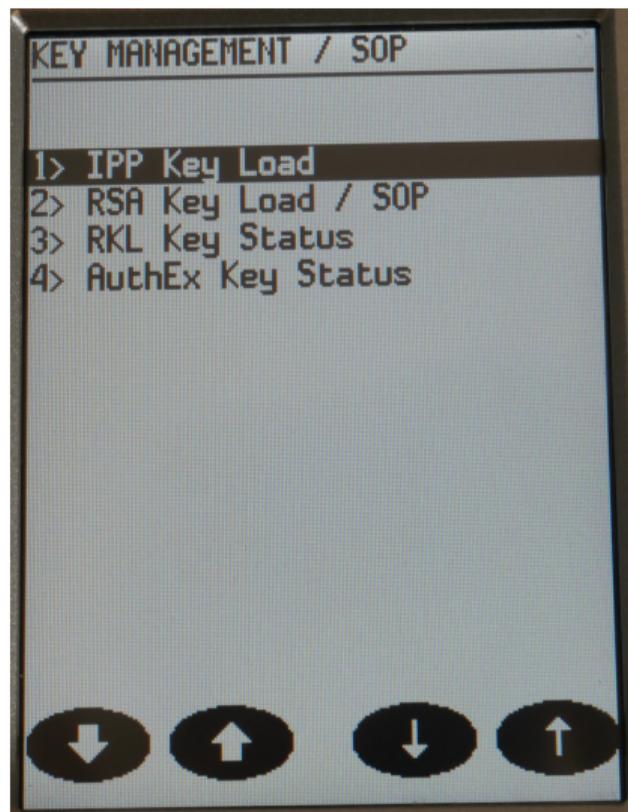
Clearing the tamper flag



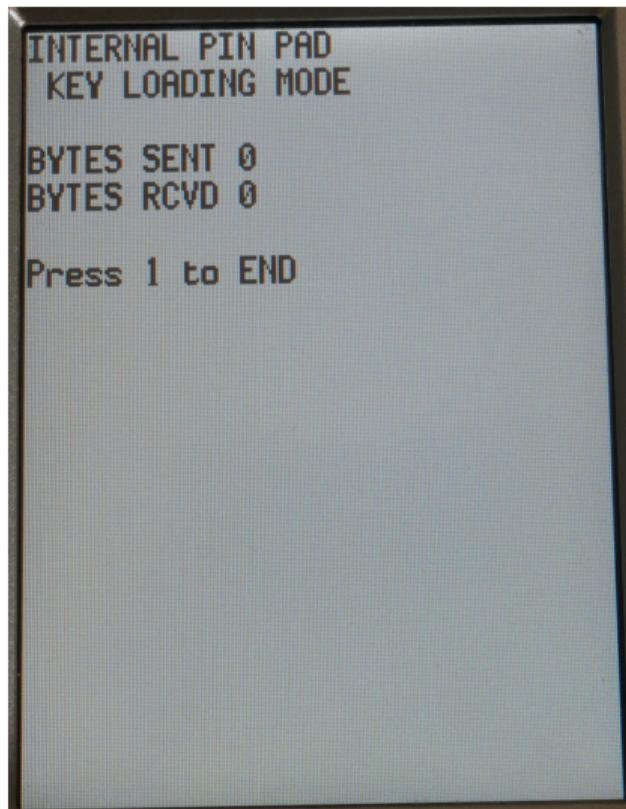
Clearing the tamper flag



Clearing the tamper flag



Clearing the tamper flag



Clearing the tamper flag

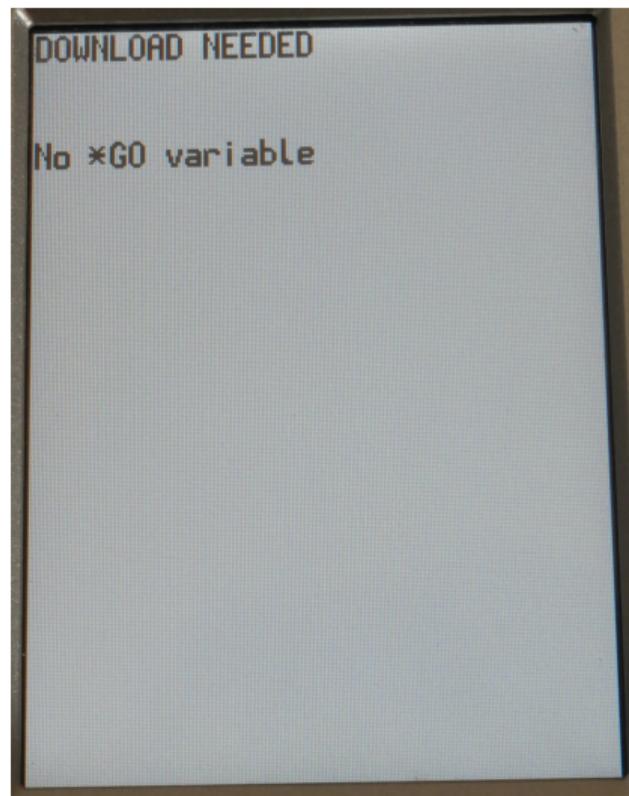
- ▶ Press '2' on IPP Key Load screen

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:

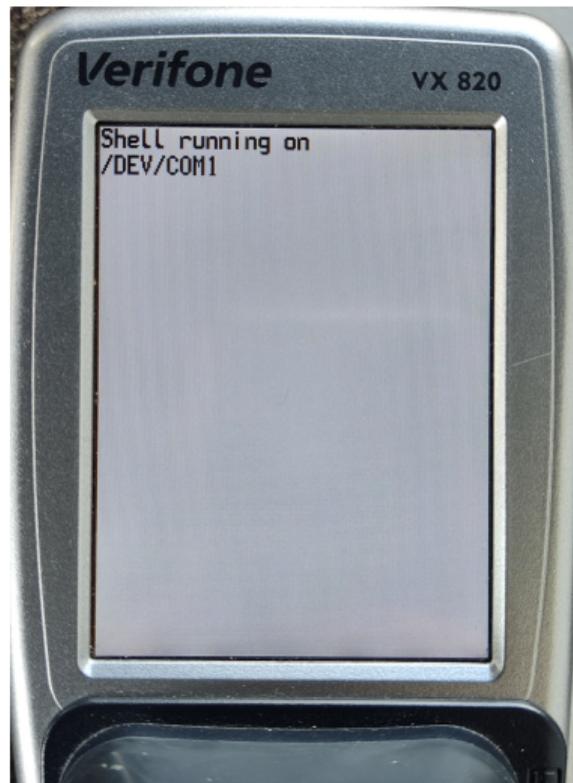
Source: research by @ivachou and @A1ex_S:
<https://www.paymentvillage.org/resources>

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!**

```
==== Authenticated 0x77266 bytes in 75 mili seconds
==== Loaded VX Module   ==== SBI V. 03_04 (Jan 13 20
==== SEARCHING USB STICK
==== USB NOT FOUND
==== LOADING FROM NAND
==== Read from nand 0x772a6 bytes in 63 mili seconds
==== Vx File Auth using PedGuard

==== Authenticated 0x77266 bytes in 75 mili seconds
==== Loaded VX Module
```

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	
00000000:	8fc3 9ba1	800 0000 3000 f245 0000 00000..E....
00000010:	0400 0000 0300 0000	f897 1800 0000 0000
00000020:	0000 0000 0000 0000	0000 0000 0000 0000
00000030:	0000 0000 5960 a808	0100 0000 0000 0000	...Y.....
00000040:	0000 0000 f045 0100	0100 0000 6000 0000	...E.....
00000050:	9045 0100 1049 0100	0001 0000 0000 0000	.E...I.....
00000060:	0000 0000 9ad9 f245	0000 a0e1 0000 a0e1E.....
00000070:	0000 a0e1 0000 a0e1	0000 a0e1 0000 a0e1
00000080:	0000 a0e1 0000 a0e1	0000 a0e1 0000 a0e1

```
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 :)



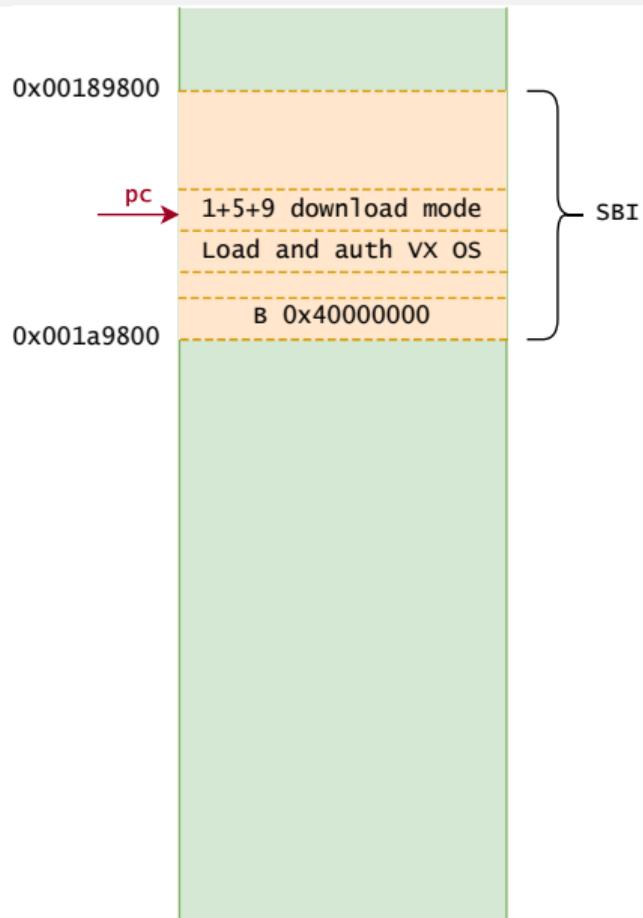
Code execution

The plan

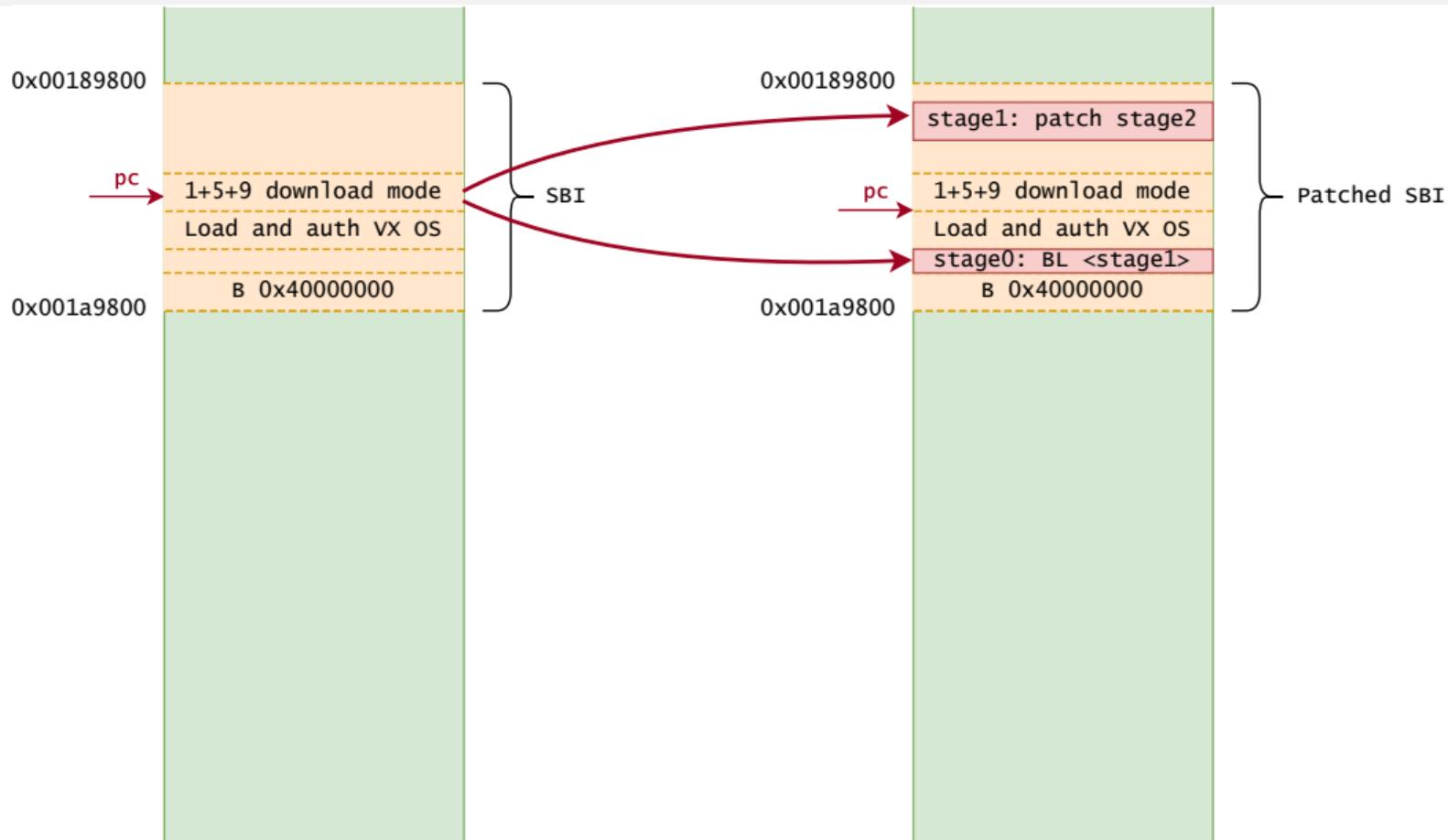
Use code execution in SBI to get control over Verix:

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.0UT");`

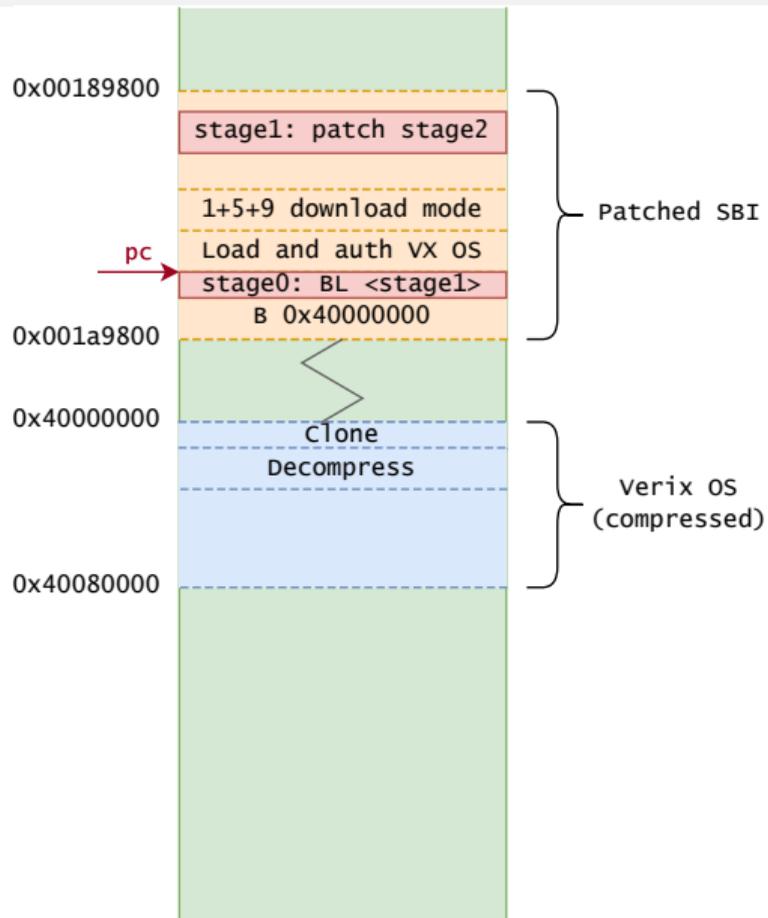
Code execution (1)



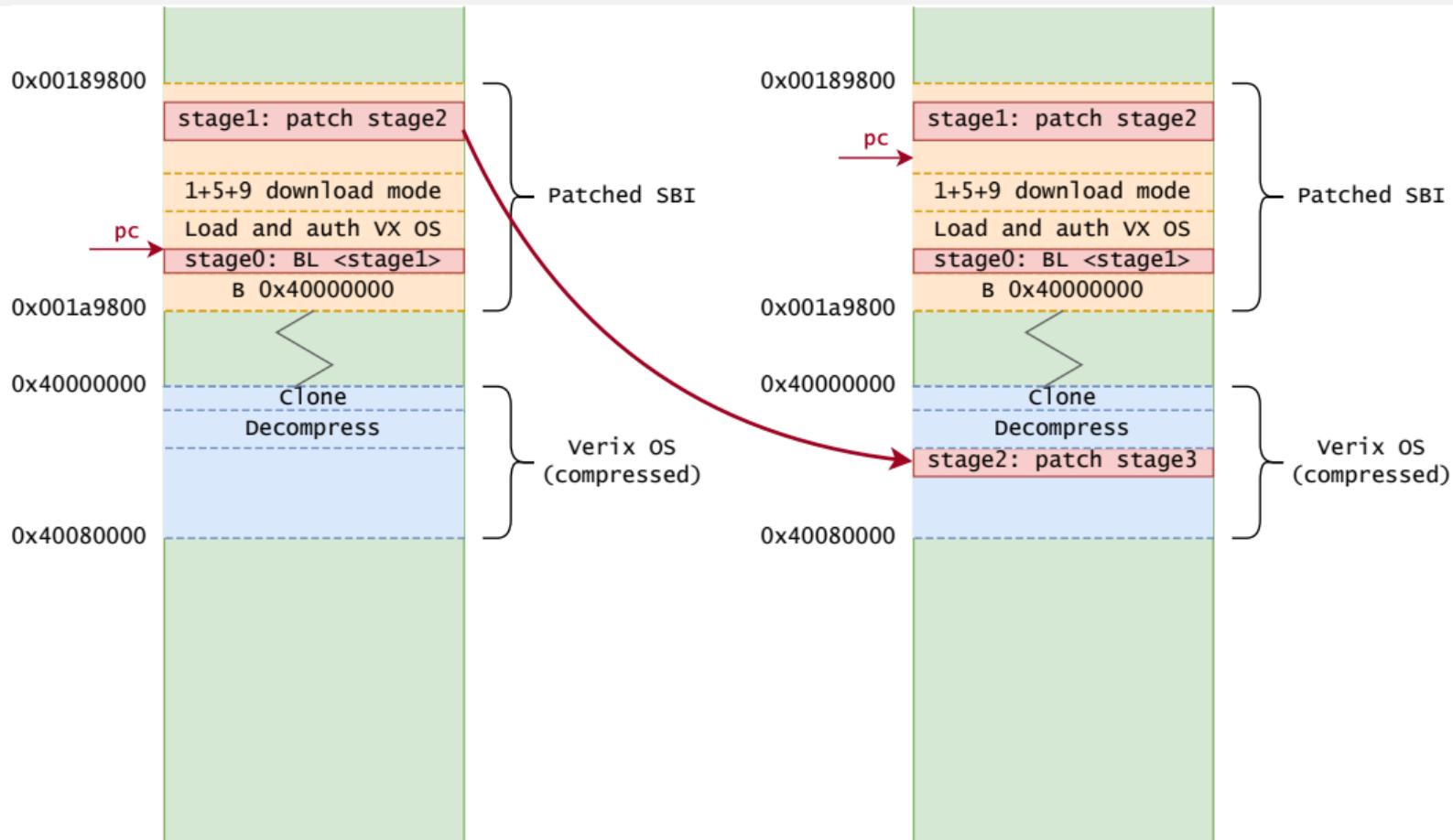
Code execution (1)



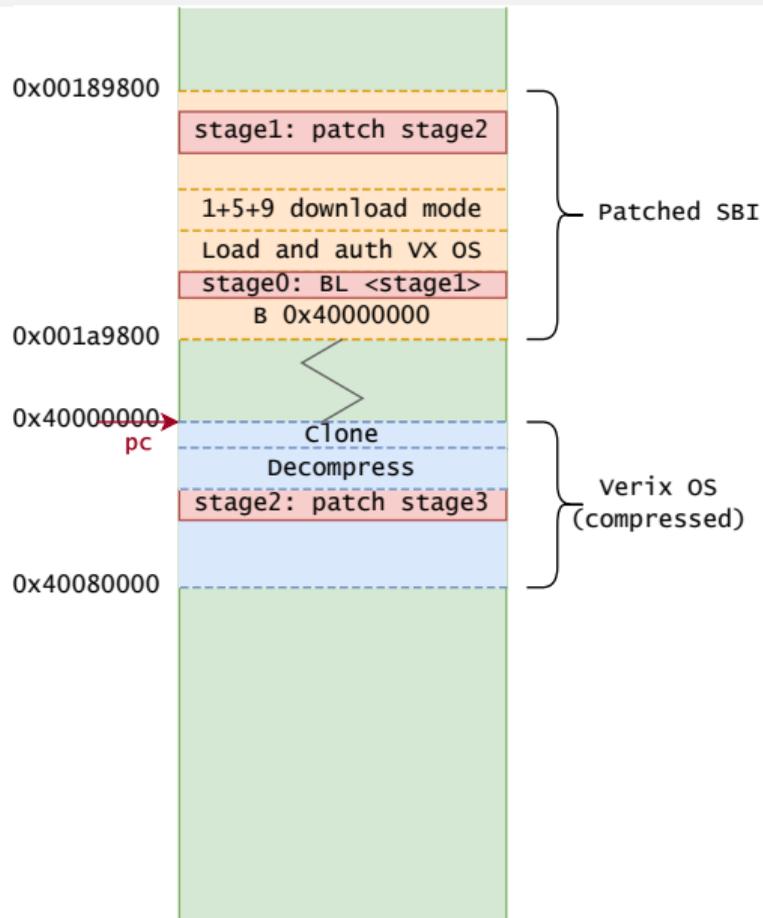
Code execution (2)



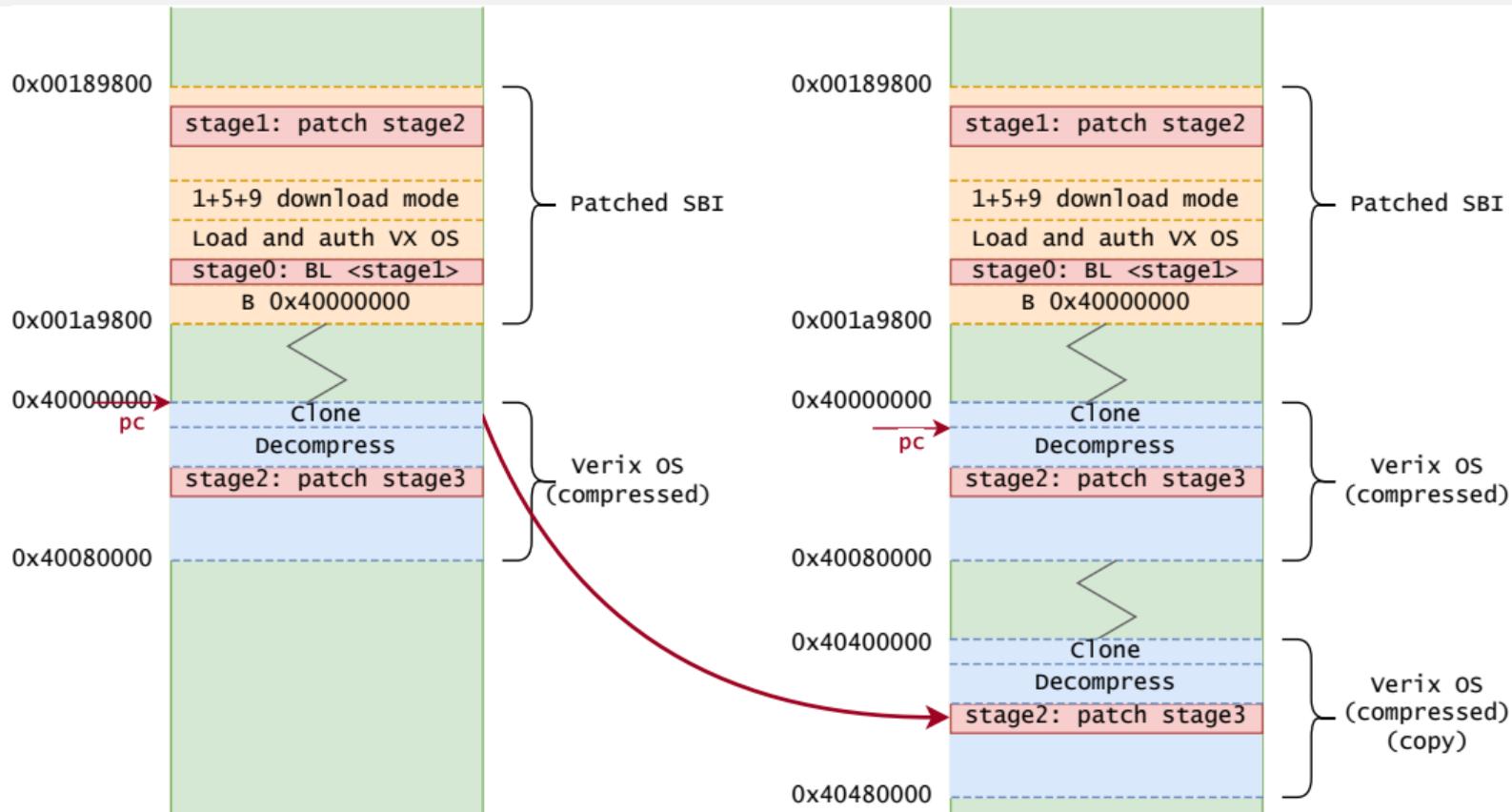
Code execution (2)



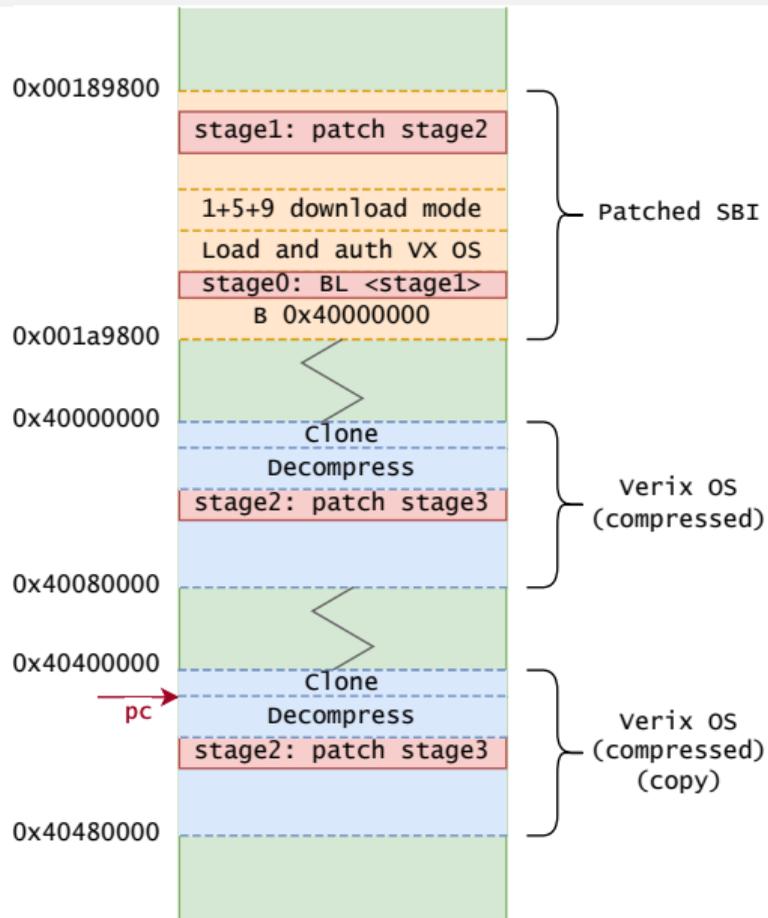
Code execution (3)



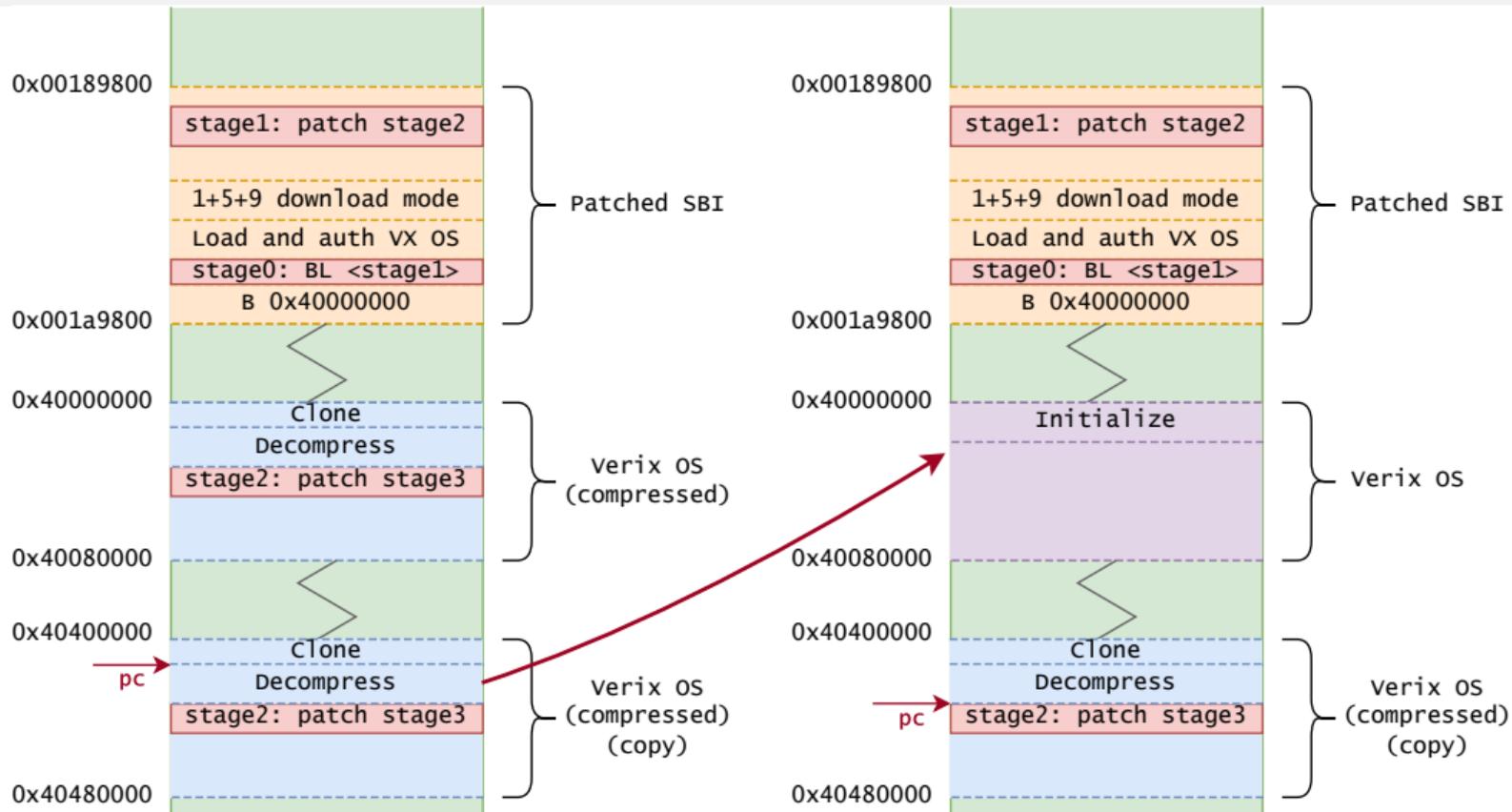
Code execution (3)



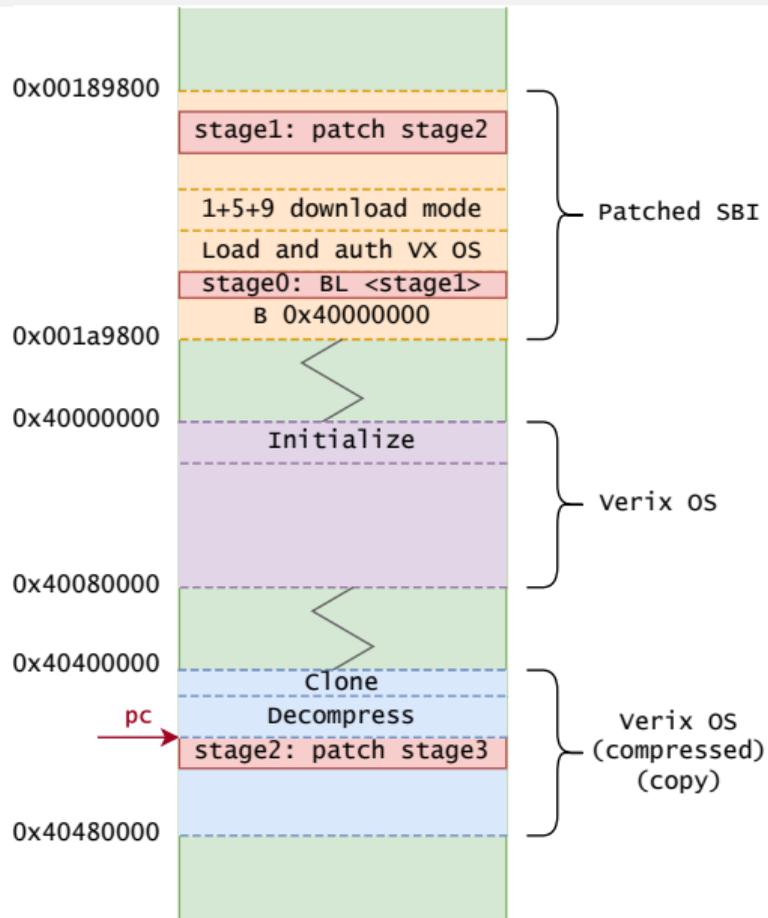
Code execution (4)



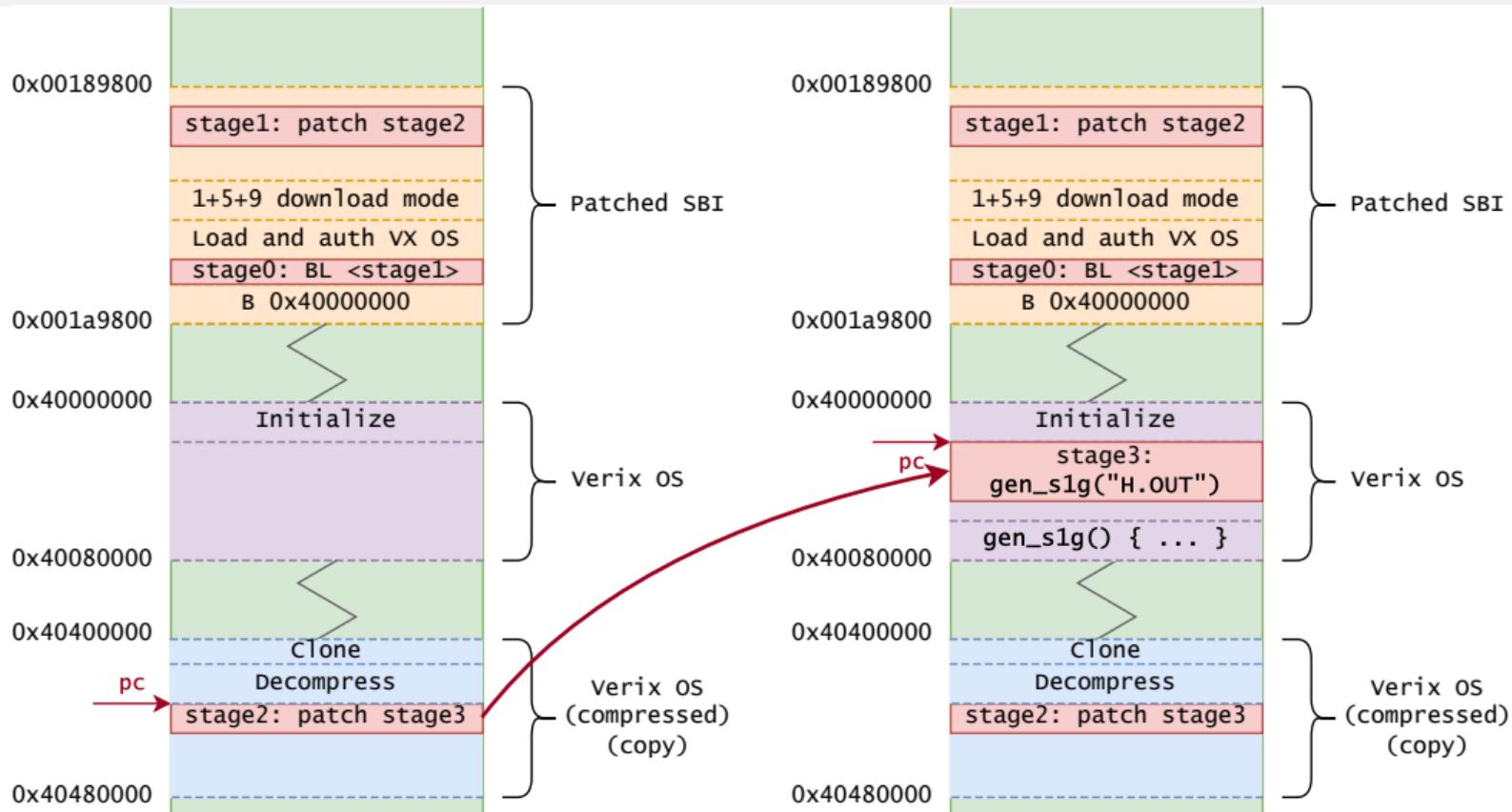
Code execution (4)



Code execution (5)

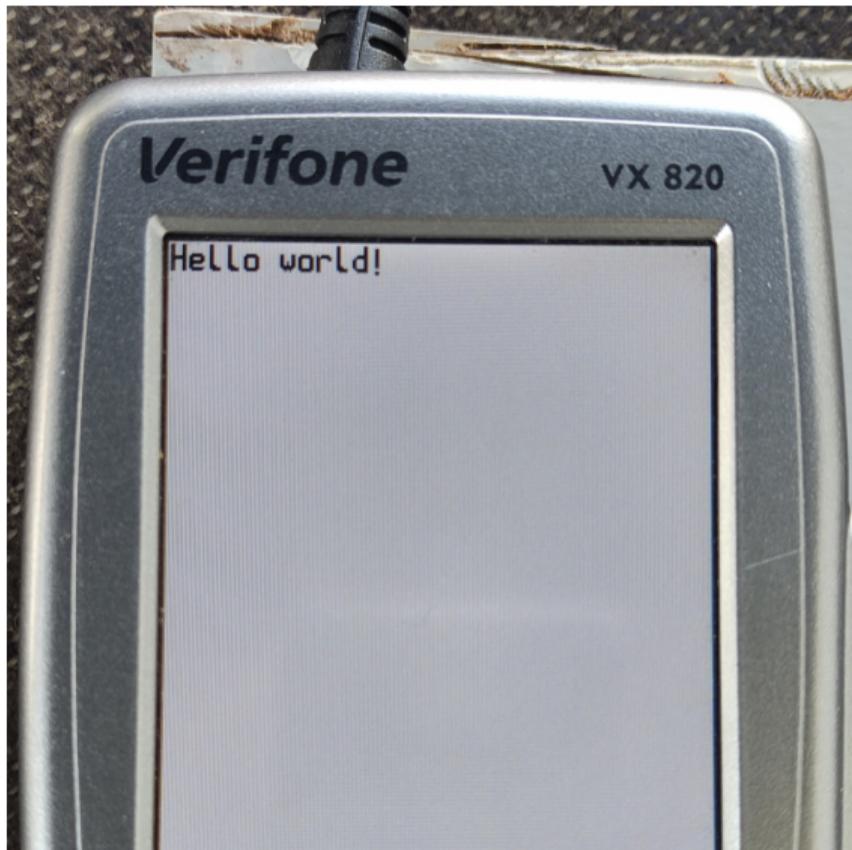


Code execution (5)



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

```
1 ENTRY(main)
2 OUTPUT_FORMAT(elf32-bigarm)
3
4 SECTIONS
5 {
6     . = 0x70420074;
7     .text : { *(.text) }
8     .data : { *(.data) }
9     .bss : { *(.bss) }
10 }
```

```
$(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. → **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/>