Progetto del corso di Sicurezza informatica e Internet A.A. 2017-2018

Andrea Graziani (0273395)[1], Alessandro Boccini (0277414)[1], and Ricardo Gamucci (0274716)[1]

[1]Università degli Studi di Roma Tor Vergata

27 marzo 2019

Indice

Capitolo 1

Analisi tecnica del malware

Secondo il rapporto stilato dalla NCCIC[1], il malware denominato **FASTCash** è composto da una serie **12 file** i quali, attraverso tecniche di **code injection** tali da alterare il normale comportamento di uno o più processi legittimi, che hanno consentito l'ispezione e alterazione dei dati trasmessi durante transazioni basate su **protocollo ISO 8583**, hanno permesso agli attaccanti di eseguire operazioni di prelievo fraudolento di denaro dagli ATM. Tra questi file, di cui riportiamo una lista completa in 1.1, spiccano per importanza:

- Tre file progettati per essere eseguibili su sistemi operativi AIX, uno dei quali responsabile dell'esecuzione della code injection contro i processi operanti sul server bersaglio. Due di essi sono stati analizzati nelle sezioni 1.2 e 1.1
- Due versioni di un malware capace di modificare le impostazioni del firewall.
- Un **trojan** capace di consentire **accesso remoto completo** al sistema bersaglio.

Prima di procedere con la descrizione dettagliata di alcuni dei file che compongono il malware, riportiamo di seguito i vari tool utilizzati durante le nostre analisi:

`strings` [2] Usato per l'estrazione di tutte le stringhe stampabili contenuti in un file.

`stat` [3] Utilizzato per ottenere alcune informazioni di base dei file tra cui nome, dimensione, data di ultima modifica, ecc.

`file` [4] Usato per determinare la tipologia di appartenenza di uno specifico file.

`onlinedisassembler` [5] Il de-assemblaggio dei file è stato eseguito utilizzando il servizio cloud **onlinedisassembler** che ci ha permesso di ricavare

[1] Cfr. `https://www.us-cert.gov/ncas/analysis-reports/AR18-275A`
[2] Cfr. `https://linux.die.net/man/1/strings`
[3] Cfr. `https://linux.die.net/man/1/stat`
[4] Cfr. `https://linux.die.net/man/1/file`
[5] Cfr. `https://onlinedisassembler.com/`

facilmente i listati di codice assembly dei file scritti per le architetture PowerPC™.

Tabella 1.1: Lista dei file del malware FASTCash

Nome file	**SHA256 digest**
Lost_File.so	10ac312c8dd02e417dd24d53c99525c29d74dcbc84730351ad7a4e0a4b1a0eba
Unpacked_dump_4a740227eeb82c20...	10ac312c8dd02e417dd24d53c99525c29d74dcbc84730351ad7a4e0a4b1a0eba
Lost_File1_so_file	3a5ba44f140821849de2d82d5a137c3bb5a736130dddb86b296d94e6b421594c
4f67f3e4a7509af1b2b1c6180a03b3...	4a740227eeb82c20286d9c112ef95f0c1380d0e90ffb39fc75c8456db4f60756
5cfa1c2cb430bec721063e3e2d144f...	820ca1903a30516263d630c7c08f2b95f7b65dffceb21129c51c9e21cf9551c6
Unpacked_dump_820ca1903a305162...	9ddacbcd0700dc4b9babcd09ac1cebe23a0035099cb612e6c85ff4dffd087a26
8efaabb7b1700686efedadb7949eba...	a9bc09a17d55fc790568ac864e3885434a43c33834551e027adb1896a463aafc
d0a8e0b685c2ea775a74389973fc92...	ab88f12f0a30b4601dc26dbae57646efb77d5c6382fb25522c529437e5428629
2.so	ca9ab48d293cc84092e8db8f0ca99cb155b30c61d32a1da7cd3687de454fe86c
Injection_API_executable_e	d465637518024262c063f4a82d799a4e40ff3381014972f24ea18bc23c3b27ee
Injection_API_log_generating_s	e03dc5f1447f243cf1f305c58d95000ef4e7dbcc5c4e91154daa5acd83fea9a8
inject_api	f3e521996c85c0cdb2bfb3a0fd91eb03e25ba6feef2ba3a1da844f1b17278dd2

1.1 Il file `Injection_API_executable_e`

L'output ottenuto dal tool `file` indica che il file `Injection_API_executable_e`, di cui abbiamo riportato alcuni dettagli nella tabella 1.2, è un **eseguibile** di tipo **eXtended COFF** (**XCOFF**), ovvero una versione migliorata ed estesa del formato **Common Object File Format** (**COFF**), il formato standard per la definizione dei file a livello strutturale nei sistemi operativi UNIX[6] fino al 1999[7], anno della definitiva adozione dello standard **Executable and Linkable Format** o **ELF**.

Il formato XCOFF è uno standard proprietario sviluppato da IBM[8] ed adottato nei sistemi operativi **Advanced Interactive eXecutive** o **AIX**, una famiglia di sistemi operativi proprietari basati su Unix sviluppati dalla stessa IBM.[9]

In questa sezione cercheremo di descrivere le capacità di questo file tra cui, come già detto precedentemente, quella di eseguire tecniche di **code injection** a danno di un processo legittimo con l'intento di alterarne il comportamento a favore degli attaccanti.

1.1.1 Analisi delle stringhe

Cominciamo l'analisi studiando alcune delle stringhe più importanti che è possibile estrarre ricorrendo al tool `strings`.

Osservando innanzitutto il formato della directory di installazione predefinita delle librerie del compilatore **GCC** nei sistemi operativi AIX, riportato

[6] Cfr. `https://it.wikipedia.org/wiki/COFF`
[7] Cfr. `https://en.wikipedia.org/wiki/Executable_and_Linkable_Format`
[8] Cfr. `https://www.ibm.com/support/knowledgecenter/ssw_aix_72/com.ibm.aix.files/XCOFF.htm`
[9] Cfr. `https://www.ibm.com/it-infrastructure/power/os/aix`

Tabella 1.2: Dettagli del file `Injection_API_executable_e`

Descrizione	**Valore**
Nome	`Injection_API_executable_e`
Dimensione (*byte*)	`89088`
Data ultima modifca	`2018-11-09 11:08:40.000000000 +0100`
Tipo di file	`64-bit XCOFF executable or object module`
MD5 digest	`b3efec620885e6cf5b60f72e66d908a9`
SHA1 digest	`274b0bccb1bfc2731d86782de7babdeece379cf4`
SHA256 digest	`d465637518024262c063f4a82d799a4e40ff3381014972f24ea18bc23c3b27ee`
SHA512 digest	`a36dab1a1bc194b8acc220b23a6e36438d43fc7ac06840daa3d010fddcd9c316 8a6bf314ee13b58163967ab97a91224bfc6ba482466a9515de537d5d1fa6c5f9`

per comodità nel listato 1.1[10], possiamo facilmente conoscere dal frammento mostrato nel listato 1.1 sia la versione di GCC che quella del sistema operativo AIX utilizzati per eseguire la *build* del malware, le quali risultano essere pari a 4.8.5[11] e 7.1[12] rispettivamente. Dallo stesso listato si può apprendere inoltre l'architettura del sistema: la PowerPC™.

Listing 1.1: Formato della directory di installazione predefinita delle librerie GCC nei sistemi operativi AIX

```
/opt/freeware/lib/gcc/<architecture_AIX_level>/<GCC_Level>
```

Listing 1.2: Stringhe estratte dal file `Injection_API_executable_e` (1)

```
347 ...
348 /opt/freeware/lib/gcc/powerpc-ibm-aix7.1.0.0/4.8.5/ppc64:/
    opt/freeware/lib/gcc/powerpc-ibm-aix7.1.0.0/4.8.5:/opt/
    freeware/lib/gcc/powerpc-ibm-aix7.1.0.0/4.8.5/../../..:/
    usr/lib:/lib
349 ...
```

Sfortunatamente non è stato possibile risalire alla versione degli aggiornamenti, identificati dalla stessa IBM con il nome di *Technology Levels* (TLs) [13], installati sul sistema operativo bersaglio al momento dell'attacco, pertanto non possiamo escludere lo sfruttamento di una qualche vulnerabilità nota da parte degli attaccanti; in ogni caso, dal momento che il malware è stato compilato per la versione 7.1 di AIX, possiamo presupporre che la versione del sistema operativo attaccato fosse almeno pari alla 7.1. E' importante ricordare che il supporto ufficiale da parte di IBM nei confronti della versione 7.1 di AIX TL0, sostituita dalla ben più moderna versione 7.2 rilasciata nel dicembre 2015, è stata già terminata nel novembre 2013, benché la versione 7.1 TL5 riceverà ancora aggiornamenti da parte di IBM fino ad aprile 2022.[14]

Listing 1.3: Stringhe estratte dal file `Injection_API_executable_e` (2)

[10] Cfr: `http://www.perzl.org/aix/index.php%3Fn%3DMain.GCCBinariesVersionNeutral`
[11] Ulteriori dettagli su: `https://gcc.gnu.org/gcc-4.8/`
[12] Ulteriori dettagli su: `https://www-01.ibm.com/support/docview.wss?uid=isg3T1012517`
[13] Cfr: `http://ibmsystemsmag.com/aix/tipstechniques/migration/oslevel_versions/`
[14] Cfr: `https://www-01.ibm.com/support/docview.wss?uid=isg3T1012517`

```
944 ...
945 IBM XL C for AIX, Version 11.1.0.1
946 ...
```

Il particolare mostrato nel listato 1.3 dimostra l'uso da parte degli attaccanti del software **XL C/C++ for AIX** versione 11.1.0.1, un compilatore C/C++ appositamente ottimizzato dalla IBM per i propri sistemi operativi[15], confermando, insieme ai numerosi riferimenti alle ben note librerie standard di C, come il C/C++ sia stato il linguaggio di programmazione scelto per implementare il malware.

Osservando il frammento mostrato nel listato 1.4, è possibile notare un insieme di stringhe, aventi formato `([FUNCTION_NAME] info)`, le quali, come avremo modo di notare durante l'analisi del codice assembly, fanno parte certamente di un meccanismo di logging sfruttato dagli attaccanti; tale aspetto è stato confermato dalla già citata analisi della NCCIC

Con ogni probabilità, le suddette stampe sono state realizzate per mezzo della funzione della libreria standard `snprintf`, come dimostrato dal codice assembly e dai numerosi riferimenti alla suddetta funzione presenti nel file. E' interessante notare come molte delle stampe coinvolgano numeri interi senza segno in forma esadecimale, come dimostrato dall'uso dei *conversion specifier* (le speciali sequenze di caratteri usati abitualmente nella definizione del formato di output nelle funzioni `printf`) nella forma `%llX`[16].

Queste stampe di log coinvolgono gran parte delle funzioni implementate nel file e presumibilmente sono state utilizzate dagli attaccanti per motivi di debug e racconta di informazioni arricchite anche da indicazioni temporali, come dimostrano l'uso delle funzioni `gettimeofday` e `localtime`. Inoltre, la presenza di procedura denominata `out_log`, analizzata in dettaglio in 1.2, dimostra che le suddette stampe siano state scritte in memoria di massa.

Infine le righe le righe 333, 334 e 335 del listato 1.4 indicano che il malware sia stato implementato sotto forma di una **command-line utility interattiva**; tale supposizione è stata confermata anche dall'analisi NCCIC.

Listing 1.4: Stringhe estratte dal file `Injection_API_executable_e` (3)

```
320 ...
321 [main] Inject Start
322 [main] SAVE REGISTRY
323 [main] proc_readmemory fail
324 [main] toc=%llX
325 [main] path::%s
326 [main] data(%p)::%s
327 [main] Exec func(%llX) OK
328 [main] Exec func(%llX) fail ret=%X
329 [main] Inject OK(%llX)
330 [main] Inject fail ret=%llX
331 [main] Eject OK
332 [main] Eject fail ret=%llX
333 Usage: injection pid dll_path mode [handle func toc]
334        mode = 0 => Injection
```

[15] Cfr. https://www.ibm.com/it-it/marketplace/xl-cpp-aix-compiler-power
[16] Cfr. http://man7.org/linux/man-pages/man3/printf.3.html

```
335         mode = 1 => Ejection
336 [main] handle=%llX, func=%llX, toc=%llX
337 [main] ERROR::g_pid(%X) <= 0
338 [main] ERROR::load_config fail
339 [main] ERROR::eject & argc != 7
340 [main] ERROR::g_dl_handle(%llX) <= 0
341 [main] WARNING::func_addr(%llX), toc_addr(%llX)
342 ...
```

Prima di analizzare nel dettaglio l'attacco di code injection vero e proprio, è indispensabile dapprima comprendere come vengono rappresentati e gestiti i **processi** nei sistemi operativi AIX. Ogni particolare aspetto di un processo, come, ad esempio, il suo stato, i sui livelli di privilegio o il proprio spazio di indirizzamento, è descritto da un insieme di file. Quest'ultimi, dato un processo il cui identificatore sia pari a `pid`, sono tutti raccolti nella directory `/proc/pid`. Tale sistema di gestione dei processi adottato da AIX permette di:

- Conoscere i `pid` di **tutti** i processi del sistema attraverso il listing nella directory `/proc`.
- Accedere alle informazioni di un dato processo attraverso semplici operazioni di lettura e scrittura sui suddetti file, utilizzando ad esempio le *system call* standard come `open()`, `close()`, `read()` e `write()`. [17]

Di questi file, alcuni dei quali sono riportati a titolo di esempio nella tabella 1.3[18], ricordiamo in particolare:

/proc/pid/as Contiene l'immagine dello spazio degli indirizzi del processo e può essere aperto sia per la lettura che per la scrittura e supporta la subroutine `lseek` per accedere all'indirizzo virtuale di interesse.[19]

/proc/pid/ctl Un file di sola scrittura attraverso cui è possibile modificare lo stato del processo e alterare dunque il suo comportamento. La scrittura avviene per mezzo di opportuni **messaggi** scritti direttamente sul file con effetti immediati.[20]

/proc/pid/status Contiene informazioni sullo stato del processo. [21]

Come mostrato nel listato 1.5, sono state individuate all'interno del file tre stringhe che fanno riferimento ai suddetti file descrittori di processo ed, in particolare, ai file `ctl`, `status` e `as`.

Come dimostrato dall'analisi della NCCIC, dalla nostra analisi del codice assembler e anche dalla presenza del *conversion specifier* `%d`, non c'è dubbio che il malware, dopo aver individuato l'identificatore del processo bersaglio, ricostruisca, per mezzo della funzione `sprintf`, i percorsi completi verso i suddetti file per poi ispezionare e manipolarne il contenuto.

[17] Cfr. IBM - *AIX Version 7.1: Files References* - pag. 232-246
[18] La lista completa è disponibile in *ivi* pag. 246
[19] *Cfr. ivi* pag. 232
[20] *Cfr. ivi* pag. 232
[21] *Cfr. ivi* pag. 232

Tabella 1.3: Sottoinsieme dei file contenuti in `/proc/pid`

File	Descrizione
`/proc/pid/status`	*Status of process* `pid`
`/proc/pid/ctl`	*Control file for process* `pid`
`/proc/pid/as`	*Address space of process* `pid`
`/proc/pid/cred`	*Credentials information for process* `pid`
`/proc/pid/sigact`	*Signal actions for process* `pid`
`/proc/pid/sysent`	*System call information for process* `pid`

Listing 1.5: Stringhe estratte dal file `Injection_API_executable_e` (4)

```
320 /proc/%d/ctl
321 /proc/%d/status
322 /proc/%d/as
```

Analizziamo ora nel dettaglio cosa può essere effettivamente scritto all'interno dei suddetti file.

La documentazione ufficiale rilasciata dalla IBM riporta l'esistenza di un insieme di **messaggi strutturati**[22], ognuno dei quali identificato da un codice operativo, rappresentato da un valore `int`, e da una serie di argomenti (se presenti)[23]. Come già detto, questi messaggi possono essere scritti direttamente nel file `ctl` di un dato processo, alterandone lo stato.

Osservando il listato 1.6, notiamo una stampa del logger all'interno è presente la stringa **PCWSTOP**; **PCWSTOP** è il nome di un messaggio definito nei sistemi operativi AIX che viene usato per sospendere l'esecuzione di un processo il cui `pid` viene passato come argomento.[24] I risultati della NCCIC e le nostre analisi sul codice assembly indicano che il malware usi questo ed altri messaggi per interrompere dapprima il processo bersaglio, accedere al suo spazio di indirizzamento, effettuare la code injection per poi riavviare il processo affinché esegua effettivamente il codice malevolo.

Listing 1.6: Stringhe estratte dal file `Injection_API_executable_e` (5)

```
319 ...
320 [proc_wait] PCWSTOP pid=%d, ret=%d, err=%d(%s)
321 [proc_wait] tid=%d, why=%d, what=%d, flag=%d, sig=%d
322 ...
```

Gli altri tipi di messaggi usati dagli attaccanti sono visibili nel listato 1.7 tra cui spiccano per importanza:

PCSET Serve per passare una serie di flag ad un processo (PR_ASYNC, PR_FORK, PR_KLC ecc.) per modificarne lo stato.[25]

PCRUN Riesegue un thread dopo essere stato arrestato.

[22] La documentazione IBM usa in modo intercambiabile il termine *messaggio* e quello di *segnale*

[23] *Cfr. ivi* pag. 242

[24] Cfr. *Ibidem*

[25] *Cfr. ivi* pag. 234

PCSENTRY Il thread corrente viene interrotto nel momento in cui richiama una specifica system call.

PCSFAULT Definisce un insieme di *hardware faults* "tracciabili" nel processo. Il thread si interrompe quando si verifica una fault.[26]

Listing 1.7: Stringhe estratte dal file `Injection_API_executable_e` (6)

```
299 ...
300 [proc_attach] PCSET pid=%d, ret=%d, err=%d(%s)
301 [proc_attach] PCSTOP pid=%d, ret=%d, err=%d(%s)
302 [proc_attach] PCSTRACE pid=%d, ret=%d, err=%d(%s)
303 [proc_attach] PCSFAULT pid=%d, ret=%d, err=%d(%s)
304 [proc_attach] PCSENTRY pid=%d, ret=%d, err=%d(%s)
305 [proc_detach] PCSTRACE pid=%d, ret=%d, err=%d(%s)
306 [proc_detach] PCSFAULT pid=%d, ret=%d, err=%d(%s)
307 [proc_detach] PCSENTRY pid=%d, ret=%d, err=%d(%s)
308 [proc_detach] PCRUN pid=%d, ret=%d, err=%d(%s)
309 ...
```

Come dimostrano i log mostrati nei listati 1.8 e 1.9, il malware non si limita solo alla scrittura dei messaggi nei file di controllo dei processi ma raccoglie ed altera le informazioni presenti nei registri del processore, parte dei quali sono riportati nella tabella 1.4[27]

Listing 1.8: Stringhe estratte dal file `Injection_API_executable_e` (7)

```
299 ...
300 [proc_getregs] GETREG pid=%d, ret=%d, err=%d(%s)
301 [proc_getregs] GETSTATUS pr_syscall=%d, pr_why=%d, pr_what=%
    d, pr_flags=%d, pr_cursig=%d
302 [proc_setregs] SETREG pid=%d, ret=%d, err=%d(%s)
303 ...
```

Listing 1.9: Stringhe estratte dal file `Injection_API_executable_e` (8)

```
320 [out_regs] IAR=%llX
321 [out_regs] MSR=%llX
322 [out_regs] CR=%llX
323 [out_regs] LR=%llX
324 [out_regs] CTR=%llX
325 [out_regs] GPR%d=%llX
```

Mostriamo infine nel listato 1.10 un log che dimostra come il malware dapprima accede ispezionando l'area di memoria riservata di un processo per poi alterarla eseguendo un'operazione di scrittura, completando in tal modo l'attacco di code injection che si conclude definitamente con il riavvio del processo attaccato.

Listing 1.10: Stringhe estratte dal file `Injection_API_executable_e` (9)

[26] *Cfr. ibidem*

[27] Cfr. *AIX Version 7.1: Assembler Language Reference* per una lista completa oppure visita `https://www.ibm.com/support/knowledgecenter/en/ssw_aix_71/com.ibm.aix.alangref/idalangref_arch_overview.htm` o `https://www.ibm.com/support/knowledgecenter/en/ssw_aix_71/com.ibm.aix.kdb/kdb_registers.htm`

Tabella 1.4: Breve descrizione dei registri ispezionati dal malware

Registro	Nome esteso	Descrizione
LR	Link Register	E' usato per ospitare l'indirizzo dell'istruzione successiva ad una operazione di salto. E' usata principalmente per ospitare l'indirizzo di ritorno al termine di una funzione.
CR	Condition Register	Un registro da 32 bit usato per specificare varie classi di operazioni.
CTR	Control Register	Un registro da 32 bit usato per specificare varie classi di operazioni.
IAR	Instruction Address Register	Usato per contenere l'indirizzo dell'istruzione successiva.
MSR	Machine State Register	Registro da 32 bit usato per specificare varie classi di operazioni.
r0-r31	General Purpose Registers (GPRs) from 0 through 31	Registri per usi generici.

```
308 ...
309 [proc_readmemory] ret=%d, err=%d(%s), addr=%p, len=%d, data
    =%p
310 [proc_readmemory] (%X~%X) %02X %02X %02X %02X %02X %02X %02X
     %02X %02X %02X %02X %02X %02X %02X %02X %02X %02X
311 [proc_writememory] ret=%d, err=%d(%s), addr=%p, len=%d, data
    =%p
312 ...
```

1.1.2 Analisi del codice assembly

Tabella 1.5: Alcune istruzioni assembly disponibili nell'architettura PowerPC^{TM}

Istruzione	Nome	Argomenti	Descrizione
`bl`	*Branch Link*	*`target_address`*	*Branches to a specified target address.*
`mfcr`	*Move From Condition Register*	`RT`	*Copies the contents of the Condition Register into a general-purpose register.*
`std`	*STore Doubleword*	`RS,`*`Offset`*`,RSML`	*Store a doubleword of data from a general purpose register into a specified memory location.*
`stw`	*STore Word*	`RS,`*`Offset`*`,RSML`	*Stores a word of data from a general-purpose register into a specified location in memory.*
`li`	*Load Immediate*	`RT,`*`Value`*	*Copies specified value into a general-purpose register.*
`ld`	*Load Doubleword*	`RT,`*`Offset`*`,RS`	*Load a doubleword of data into the specified general purpose register.*
`mr`	*Move Register*	`RT,RS`	*Copies the contents of one register into another register.*
`addi`	*ADD Immediate*	`RT,RS,`*`Value`*	*Place the sum of the contents of RA and the 16-bit two's complement integer value, sign-extended to 32 bits, into the target RT.*
`mtrl`	*Move To Link Register*	`RS`	*Copies the contents of RS register into Link Register.*
`extsw`	*Extend Sign Word*	`RT,RS`	*Copy the low-order 32 bits of a general purpose register into another general purpose register, and signextend the fullword to a doubleword in size (64 bits).*

1.1.2.1 La procedura `main`

La parte iniziale della procedura `main` è caratterizzata da una serie di operazioni che coinvolgono stringhe come dimostrano la serie di istruzioni di salto condizionato verso le funzioni `strlen` (riga 7028), `strncpy` (riga 7035) e `strtoull` (riga 6973, 6984 e 6995) che probabilmente sono state utilizzate per raccogliere informazioni. Sono presenti due istruzioni di salto vero le funzioni `atoi` (riga 7015 e 7041) usate per convertire i parametri passati dagli attaccanti attraverso la linea di comando il che dimostra la natura interattiva del malware.

Dopo una serie di istruzioni di salto verso procedure varie procedure di inizilizzazione, tra cui spiccano `load_config` e `get_func_addr`, viene raggiunta la porzione di codice mostrata nel listato 1.11 dove, dopo aver copiato i dati necessari in alcuni registri attraverso le apposite istruzioni mr, vengono eseguite a cascata due istruzioni di salto verso una procedura chiamata inject, che contiene il codice operativo per l'esecuzione della code injection.

Listing 1.11: Codice assembly estratto dal file `Injection_API_executable_e`

```
308 mr      r3,r10
309 mr      r4,r9
310 addi    r5,r2,-728
311 bl      0x1000256c <.inject>
312 li      r3,120
313 bl      0x10003468 <.sleep>
314 ld      r2,40(r1)
315 addi    r10,r31,120
316 lwz     r9,116(r31)
317 extsw   r9,r9
318 mr      r3,r10
319 mr      r4,r9
320 li      r5,0
321 bl      0x1000256c <.inject>
322 bl      0x10001154 <.CloseHandle>
```

1.1.2.2 La procedura `inject`

In questo paragrafo descriveremo le operazioni eseguite dalla procedura chiamata `inject` a cui gli autori del malware hanno affidato il compito di eseguire l'attacco di code injection a danno del processo bersaglio. Nel listato 1.12 viene mostrato il frammento corrispondente alla prima parte della suddetta funzione, all'interno del quale possiamo distinguere le seguenti operazioni:

- Copia nel *link register* dell'indirizzo di ritorno dal registro `r0` attraverso l'istruzione `mflr`.
- Inizializzazione di vari registri necessari attraverso varie istruzioni `std` e `mr` che coinvolgono i registri `r4`, `r5`,`r9` e `r31`, quest'ultimo probabilmente adibito ad ospitare l'indirizzo di memoria di base da cui viene computato l'indirizzo da cui prelevare i dati dalla memoria. Si presupponi che i registri precedentemente menzionati ospiteranno gli indirizzi alle celle di memoria contenenti il codice malevolo che verrà poi scritto all'interno dello spazio di indirizzamento del processo bersaglio.
- Esecuzione della code injection vera e propria attraverso una serie di istruzioni di salto (`bl`) verso l'indirizzo `0x10002f00` corrispondente alla prima istruzione della funzione `memset` preceduta dalle necessarie inizializzazioni dei registri di input attraverso varie istruzioni `mr`.

Listing 1.12: Codice assembly estratto dal file `Injection_API_executable_e`

```
308 mflr    r0
309 std     r0,16(r1)
310 std     r31,-8(r1)
311 stdu    r1,-1520(r1)
312 mr      r31,r1
313 std     r3,1568(r31)
314 mr      r9,r4
315 std     r5,1584(r31)
316 stw     r9,1576(r31)
```

```
317 li       r9,0
318 stw      r9,120(r31)
319 li       r9,0
320 std      r9,144(r31)
321 addi     r10,r31,152
322 li       r9,384
323 mr       r3,r10
324 li       r4,0
325 mr       r5,r9
326 bl       0x10002f00 <.memset>
327 nop
328 addi     r10,r31,536
329 li       r9,384
330 mr       r3,r10
331 li       r4,0
332 mr       r5,r9
333 bl       0x10002f00 <.memset>
334 nop
335 addi     r10,r31,920
336 li       r9,256
337 mr       r3,r10
338 li       r4,0
339 mr       r5,r9
340 bl       0x10002f00 <.memset>
```

Dopo una serie di istruzioni di salto verso la funzione `memset`, ed aver dunque conclusa le operazioni di modifica della memoria del processo attaccato, possiamo osservare le successive operazioni eseguite dal listato 1.13 in cui apprendiamo che:

- Vengono eseguite ben tre istruzioni `bl` per permettere l'esecuzione della procedura `out_log` per effettuare la scrittura delle informazioni di interesse su un file esterno.
- Vengono diverse istruzioni di salto per eseguire varie procedure tra cui quella denominata `proc_attach`, usata probabilmente per modificare alcune informazioni di stato del processo, la `proc_wait`, usata probabilmente per arrestare l'esecuzione del processo bersaglio, `proc_getregs` ed `out_regs` usate rispettivamente per leggere i valori contenuti nei registri e successivamente scriverli in un file di log.

Listing 1.13: Codice assembly estratto dal file `Injection_API_executable_e`

```
308 bl       0x10000674 <.out_log>
309 bl       0x10001220 <.proc_attach>
310 li       r3,0
311 bl       0x10001a28 <.proc_continue>
312 li       r3,0
313 li       r4,0
314 bl       0x10001b44 <.proc_wait>
315 ld       r3,728(r2)
316 bl       0x10000674 <.out_log>
317 addi     r9,r31,152
318 mr       r3,r9
```

```
319 bl       0x10001ee4 <.proc_getregs>
320 addi     r9,r31,152
321 mr       r3,r9
322 bl       0x10000c80 <.out_regs>
```

Dopo una serie di istruzioni di salto verso altre funzioni, tra cui figura una denominata `proc_readmemory`, avviene l'ultima fase della code injection durante la quale, come dimostrato dal listato 1.14, viene alterata la memoria del processo bersaglio attraverso istruzioni di salto verso le procedure `proc_writememory`, usata probabilmente per indurre il processo bersaglio a eseguire il codice malevolo copiato in precedenza, e la `proc_setregs` usata per alterare il contenuto dei registri e dunque modificare il futuro comportamento del processo. La procedura si conclude con il riavvio del processo e una lunga fase di log attraverso una grande quantità di istruzioni di salto verso la procedura `out_log`.

Listing 1.14: Codice assembly estratto dal file `Injection_API_executable_e`

```
308 bl       0x10002460 <.proc_writememory>
309 addi     r9,r31,536
310 mr       r3,r9
311 bl       0x10000c80 <.out_regs>
312 addi     r9,r31,536
313 mr       r3,r9
314 bl       0x10002068 <.proc_setregs>
315 li       r3,3
316 bl       0x10001a28 <.proc_continue>
317 li       r3,6
318 li       r4,11
319 bl       0x10001b44 <.proc_wait>
320 addi     r9,r31,536
321 mr       r3,r9
322 bl       0x10001ee4 <.proc_getregs>
323 addi     r9,r31,536
324 mr       r3,r9
325 bl       0x10000c80 <.out_regs>
```

1.1.2.3 La procedura `proc_attach`

Analizziamo nel dettaglio l'attacco al processo la quale si compone in varie fare. Nel listato possiamo osservare come vengono dapprima eseguite delle operazioni di store ward con diversi offset con un registro comune come indirizzo sorgente;

Successivamente gli attaccanti utilizzano quello che probabilmente si tratti dell'indirizzo dell'area di memoria del processo bersaglio e con ripetute operazioni si store word muove il puntatore a quell'area di memoria con step da 4 byte. Alla fine, raggiunta la posizione desiderata, sposta il risultato in vari registri e esegue un'operazione di salto (bl) che punta all'indirizzo per la funzione memset.

```
1 [
2 frame=lines,
3 caption={Codice assembly estratto dal file \texttt{Injection
    \_API\_executable\_e}},
4 label={code:AssemblyFunction-inject-04},
```

```
5  firstnumber=308]
6  <.proc_attach>:
7  mflr    r0
8  std     r0,16(r1)
9  std     r29,-24(r1)
10 std     r30,-16(r1)
11 std     r31,-8(r1)
12 stdu    r1,-352(r1)
13 mr      r31,r1
14 li      r9,0
15 stw     r9,128(r31)
16 li      r9,0
17 stw     r9,132(r31)
18 li      r9,0
19 std     r9,136(r31)
20 li      r9,0
21 std     r9,144(r31)
22 li      r9,0
23 std     r9,152(r31)
24 li      r9,0
25 std     r9,160(r31)
26 li      r9,0
27 std     r9,168(r31)
28 li      r9,0
29 stw     r9,176(r31)
30 addi    r10,r31,180
31 li      r9,140
32 mr      r3,r10
33 li      r4,0
34 mr      r5,r9
35 bl      0x10002f00 <.memset>
```

Dopo aver richiamato la funzione `memset`, certamente utilizzata dagli attaccanti per eseguire la *code injection* alterando il contenuto dello spazio di indirizzamento del processo bersaglio, la funzione `proc_attach` incomincia una fase di logging durante la quale, attraverso ripetuti salti condizionati agli indirizzi `0x100031ec`, `0x1000319c` e `0x10000674`, corrispondenti agli indirizzi delle funzioni `write`, `sterror` (utilizzata certamente dagli attaccanti per verificare l'output della funzione `write`), `log_out`, vengono archiviati in un file esterno il contenuto dei registri di interesse che paiono essere i registri r31, r30, r29 e r9 che vengono copiati con ripetute istruzioni `mr` in registri ausiliari (r4, r5, r6 e r7 rispettivamente) prima di essere inviati come input alla funzione `log_out`.

```
1  [
2  frame=lines,
3  caption={Codice assembly estratto dal file \texttt{Injection
       \_API\_executable\_e}},
4  label={code:AssemblyFunction-inject-05},
5  firstnumber=308]
6  li      r9,14
7  stw     r9,136(r31)
8  li      r9,4
9  stw     r9,140(r31)
10 addi    r9,r2,-764
```

```
lwz     r9,0(r9)
extsw   r10,r9
addi    r9,r31,136
mr      r3,r10
mr      r4,r9
li      r5,8
bl      0x100031ec <.write>
ld      r2,40(r1)
mr      r9,r3
stw     r9,128(r31)
addi    r9,r2,-768
lwz     r9,0(r9)
extsw   r29,r9
ld      r9,128(r2)
lwz     r9,0(r9)
extsw   r30,r9
ld      r9,128(r2)
lwz     r9,0(r9)
extsw   r9,r9
mr      r3,r9
bl      0x1000319c <.strerror>
ld      r2,40(r1)
mr      r9,r3
lwz     r10,128(r31)
extsw   r10,r10
ld      r3,536(r2)
mr      r4,r29
mr      r5,r10
mr      r6,r30
mr      r7,r9
bl      0x10000674 <.out_log>
```

La fase di code injection si conclude con il caricamento nel registro `r0` dell'indirizzo della funzione chiamante copiato successivamente nel link register attraverso l'istruzione `mtrl`; vengono in seguito eseguite una serie di istruzioni ld per popolare i registri r29, r30 e r31 che conterranno probabilmente i valori di ritorno della funzione per poi eseguire una istruzione `blr` (*Branch Link Register*).

```
ld      r0,16(r1)
mtlr    r0
ld      r29,-24(r1)
ld      r30,-16(r1)
ld      r31,-8(r1)
blr
```

```
bl      0x10000674 <.out_log>
bl      0x10001220 <.proc_attach>
li      r3,0
bl      0x10001a28 <.proc_continue>
li      r3,0
li      r4,0
bl      0x10001b44 <.proc_wait>
ld      r3,728(r2)
bl      0x10000674 <.out_log>
```

```
addi    r9,r31,152
mr      r3,r9
bl      0x10001ee4 <.proc_getregs>
addi    r9,r31,152
mr      r3,r9
bl      0x10000c80 <.out_regs>
addi    r8,r31,536
addi    r10,r31,152
li      r9,384
mr      r3,r8
mr      r4,r10
mr      r5,r9
bl      0x1000324c <.memmove>
nop
ld      r9,536(r31)
addi    r9,r9,-16
mr      r3,r9
li      r4,16384
bl      0x10000b48 <.file_dump>
```

1.2 Il file `2.so`

`2.so` è un file di tipo **eXtended COFF** che, come dimostreremo all'interno di questa sezione, è stato progettato per l'ispezione e la manipolazione dei dati contenuti nei messaggi basati sul protocollo **ISO8583** scambiati tra i sistemi informatici degli istituti finanziari. Come dimostrato anche dalla già citata analisi AR18-275A della NCCIC, il file, come suggerisce anche l'estensione `.so`, rappresenta una **shared library** che, esportando una grande quantità di metodi in grado di interagire con i messaggi basati sul suddetto protocollo, permette agli attaccanti di alterare le transazioni finanziare a proprio favore.

Tabella 1.6: Dettagli del file `2.s0`

Descrizione	**Valore**
Nome	`2.so`
Dimensione (*byte*)	`110592`
Data ultima modifca	`2018-11-09 11:08:40.000000000 +0100`
Tipo di file	`64-bit XCOFF executable or object module`
MD5 digest	`b66be2f7c046205b01453951c161e6cc`
SHA1 digest	`ec5784548ffb33055d224c184ab2393f47566c7a`
SHA256 digest	`ca9ab48d293cc84092e8db8f0ca99cb155b30c61d32a1da7cd3687de454fe86c`
SHA512 digest	`6890dcce36a87b4bb2d71e177f10ba27f517d1a53ab02500296f9b3aac021810 7ced483d70d757a54a5f7489106efa1c1830ef12c93a7f6f240f112c3e90efb5`

1.2.1 Analisi delle stringhe

Listing 1.15: Stringe estratte dal file `2.so` (1)

```
465 ...
466 /opt/freeware/lib/gcc/powerpc-ibm-aix6.1.0.0/4.2.0/ppc64:/
    opt/freeware/lib/gcc/powerpc-ibm-aix6.1.0.0/4.2.0:/opt/
    freeware/lib/gcc/powerpc-ibm-aix6.1.0.0/4.2.0/../../..:/
    usr/lib:/lib
467 ...
```

Seguendo lo stesso ragionamento descritto in 1.1.1, comprendiamo dal listato 1.15 che gli attaccanti si siano serviti della versione 4.2.0 di GCC[28] compatibile con l'architettura PowerPC™con sistema operativo AIX 6.1 (di quest'ultimo il supporto è terminato ufficialmente il 30 Aprile del 2017.[29]).

La libreria esporta una grandissima quantità di funzioni molte delle quali riguardanti la manipolazione dei messaggi basati su protocollo ISO8583, alcune delle quali riportate nel listato 1.16.

Listing 1.16: Stringe estratte dal file `2.so` (2)

```
545 ...
546 DL_ISO8583_MSG_Init
547 DL_ISO8583_MSG_Free
```

[28] Cfr. `http://www.gnu.org/software/gcc/gcc-4.2/`

[29] Cfr. `https://www-01.ibm.com/support/docview.wss?uid=swg21634678#AIX`

```
DL_ISO8583_MSG_SetField_Str
DL_ISO8583_MSG_SetField_Bin
DL_ISO8583_MSG_RemoveField
DL_ISO8583_MSG_HaveField
DL_ISO8583_MSG_GetField_Str
DL_ISO8583_MSG_GetField_Bin
DL_ISO8583_MSG_Pack
DL_ISO8583_MSG_Unpack
DL_ISO8583_MSG_Dump
DL_ISO8583_MSG_AllocField
DL_ISO8583_COMMON_SetHandler
DL_ISO8583_DEFS_1987_GetHandler
DL_ISO8583_DEFS_1993_GetHandler
DL_ISO8583_FIELD_Pack
DL_ISO8583_FIELD_Unpack
...
```

Listing 1.17: Stringhe estratte dal file `2.so` (3)

```
Blocked Message(msg=%04x, term=%02x, pcode=%06x, pan=%s)
Passed Message(msg=%04x, term=%02x, pcode=%06x, pan=%s)
[recv] ret=%d
send ret = %d, err = %d
/tmp/.ICE-unix/context.dat
/tmp/.ICE-unix/tmp%d_%d.log
[%04d-%02d-%02d %02d:%02d:%02d][PID:%4u][TID:%4u] %s
/tmp/.ICE-unix/config_%d
/tmp/.ICE-unix/tmprd%d_%d.log
/tmp/.ICE-unix/tmpwt%d_%d.log
[DetourInitFunc] dlopen error(%s)
[DetourInitFunc] org_func(%p) %02X %02X %02X %02X %02X %02X
    %02X %02X %02X %02X %02X %02X
[DetourInitFunc] new_func(%p) %02X %02X %02X %02X %02X %02X
    %02X %02X %02X %02X %02X %02X
[DetourInitFunc] dlsym error(%s)
Success
Failed
DetourInitFunc(%s, %s) %s
[DetourInitFunc] org_func=%p new_func=%p
[DetourAttach] hook_func_addr=%p, new_func_addr=%p
[DetourAttach] after mmap=%p
[DetourAttach] copy_func(%x) %02X %02X %02X %02X %02X %02X
    %02X %02X %02X %02X %02X %02X
[DetourAttach] hook_func_addr(%x) %02X %02X %02X %02X %02X
    %02X %02X %02X %02X %02X %02X %02X
[DetourDetach] hook_func_addr(%x) %02X %02X %02X %02X %02X
    %02X %02X %02X %02X %02X %02X %02X
```

Come dimostrato dal frammento riportato in 1.17 e dall'analisi di alcuni frammenti chiave del codice assembly in 1.2.2.5, la libreria svolge un'**intensa attività di logging** scrivendo direttamente in memoria di massa attraverso una procedura chiamata `out_dump_log`. Dal listato 1.17 possiamo osservare un'interessante riferimento al percorso `/tmp/.ICE-unix/`: in accordo alla docu-

mentazione relativa alla versione R6.8.2 di X11[30] (la famosa implementazione del X Window System[31]) il suddetto percorso viene usato per ospitare una serie di `socket` sfruttate dal protocollo **Inter-Client Exchange** (**ICE**), utilizzato per la risoluzione di varie problematiche come quelle legate all'autenticazione o al *byte order negotiation*[32]. Pertanto, sebbene ne ignoriamo le motivazioni, abbiamo motivo di ritenere che gli attaccanti abbiano interagito con la GUI session manager di X11 attraverso il protocollo ICE.

Listing 1.18: Stringhe estratte dal file `2.so` (4)

```
545 GenerateRandAmount
546 GenerateResponseTransaction1
547 GenerateResponseTransaction2
548 GenerateResponseInquiry1
549 Crypt
```

Osservando infine il frammento riportato in 1.18 possiamo comprendere l'esistenza di alcuni metodi utilizzati per rispondere alle transazioni finanziarie generate dai sistemi bancari sotto attacco.

1.2.2 Analisi del codice assembly

Benché naturalmente sprovvista di una procedura `main`, trattandosi di una shared library, il file `2.so` assume un ruolo centrale per il corretto svolgimento dell'attacco poiché esporta tutte le procedure necessarie per manipolare le transazioni elettroniche dei sistemi finanziari attaccati. I metodi esportati dal file `so.2`, una parte dei quali sono mostrati nel listato `??`, sono molto numerosi e riguardano principalmente l'ispezione e la manipolazione dei messaggi usati dal protocollo ISO8583 a cui si aggiungono altre procedure di supporto, tra cui quelle usate per implementare un meccanismo di logging (`out_dump_log`) e altre usate per gestire una tabella hash (`hashmap_new`, `hashmap_init`, `hashmap_get` ecc.). Di seguito riportiamo l'analisi di alcune delle procedure principali presenti nel file.

1.2.2.1 La procedura `DL_ISO8583_MSG_GetField_Bin`

Analizzando le prime righe di codice della procedura, riportate in parte nel listato 1.19, troviamo molte istruzioni `std` e `mr` utilizzanti i registri `r0` e `r31` come sorgenti per popolare un altro insieme di registri; perciò si può supporre che i registri `r0` e `r31` siano stati usati per contenere i dati passati come argomento alla funzione, presumibilmente l'indirizzo dell'area di memoria del messaggio da ispezionare e un riferimento al campo da estrarre (probabilmente una stringa o un identificatore numerico).

Listing 1.19: Codice assembly estratto dal file `2.so`

```
347 std     r31,-8(r1)
348 stdu    r1,-80(r1)
349 mr      r31,r1
350 mr      r0,r3
```

[30] Cfr. `https://www.x.org/releases/X11R6.8.2/doc/RELNOTES5.html`
[31] Cfr. `https://www.x.org/wiki/`
[32] Cfr. `https://www.x.org/releases/X11R7.7/doc/libICE/ICElib.html#Overview_of_ICE`

```
351 std      r4,136(r31)
352 std      r5,144(r31)
353 std      r6,152(r31)
354 stw      r0,128(r31)
```

Un altro frammento della stessa procedura, riportato nel listato 1.20, mostra come l'ispezione del campo di interesse appartenente al messaggio ISO8583 avvenga per mezzo di un ciclo; notiamo infatti diverse istruzioni di `beq` (*Branch On Equal*) aventi come argomento uno stesso indirizzo target e altrettante istruzioni `cmpdi` (*Compare Doubleword Immediate*). E' probabile che tale ciclo sia stato usato per attraversare il flusso di byte che compone un certo messaggio fino al raggiungimento dell'indirizzo corrispondente al campo di intessere che pare venga restituito alla funzione chiamante per mezzo di un'apposita istruzione `mr` coinvolgendo il registro `r3` come output (riga 14010).

Listing 1.20: Codice assembly estratto dal file `2.so`

```
347 cmpdi    cr7,r0,0
348 beq      cr7,0x10002048
349 lwz      r0,128(r31)
350 cmplwi   cr7,r0,128
351 bgt      cr7,0x10002048
352 lwz      r0,128(r31)
353 clrldi   r9,r0,32
354 ld       r11,136(r31)
355 addi     r0,r9,1
356 rldicr   r0,r0,4,59
357 add      r9,r0,r11
358 addi     r9,r9,8
359 ld       r0,0(r9)
360 cmpdi    cr7,r0,0
361 beq      cr7,0x10002048
```

1.2.2.2 La procedura `DL_ISO8583_MSG_GetField_Str`

La funzione `DL_ISO8583_MSG_GetField_Str` è sostanzialmente identica a quella precedentemente descritta sebbene il nome suggerisca che tale funzione restituisca indubbiamente una stringa piuttosto che dati binari.

1.2.2.3 La procedura `DL_ISO8583_MSG_SetField_Bin`

L'esistenza di tale procedura dimostra che il malware non si limita solo all'ispezione dei messaggi ma che è in grado di manipolarne i contenuti. Come mostrato nel listato 1.20, in modo simile alle altre procedure, il messaggio viene dapprima ispezionato per mezzo di istruzioni di salto incondizionato (b) e condizionato (ble) insieme ad opportune istruzioni di comparazione (cmplwi) fino al raggiungimento dell'indirizzo corrispondente al campo da modificare. Una successiva istruzione di salto alla procedura `DL_ISO8583_MSG_AllocField`, attraverso la quale viene presumibilmente allocata un'opportuna area di memoria atta ad ospitare il campo con i nuovi valori, è seguita infine dall'invocazione della procedura `memmove` completando così l'operazione di modifica del messaggio.

Listing 1.21: Codice assembly estratto dal file `2.so`

```
347 cmplwi  cr7,r0,128
348 ble     cr7,0x10001b84
349 li      r0,1
350 std     r0,128(r31)
351 b       0x10001c08
352 lwz     r0,208(r31)
353 clrldi  r9,r0,32
354 lwz     r0,224(r31)
355 clrldi  r0,r0,32
356 addi    r11,r31,120
357 mr      r3,r9
358 mr      r4,r0
359 ld      r5,232(r31)
360 mr      r6,r11
361 bl      0x100026e0 <._DL_ISO8583_MSG_AllocField>
362 nop
363 mr      r0,r3
364 std     r0,112(r31)
365 ld      r0,112(r31)
366 cmpdi   cr7,r0,0
367 bne     cr7,0x10001c00
368 ld      r9,120(r31)
369 lwz     r0,224(r31)
370 clrldi  r0,r0,32
371 mr      r3,r9
372 ld      r4,216(r31)
373 mr      r5,r0
374 bl      0x1000034c <.memmove>
```

1.2.2.4 La procedura `DL_ISO8583_MSG_RemoveField`

La procedura `DL_ISO8583_MSG_RemoveField` presenta a una struttura sostanzialmente identica a quelle viste finora. L'unica differenza degna di nota, come mostrato nel listato 1.22, riguarda l'eliminazione del campo specificato che avviene attraverso una chiamata alla procedura `free`.

Listing 1.22: Codice assembly estratto dal file `2.so`

```
347 mr      r3,r0
348 bl      0x100002dc <.free>
349 ld      r2,40(r1)
```

1.2.2.5 La procedura `out_dump_log`

La procedura denominata `out_dump_log`, usata, come suggerisce il nome, per ovvie finalità legate al **debugging** dell'applicazione e al **logging** di tutte le informazioni di interesse raccolte durante l'attacco, rappresenta quella invocata il maggior numero di volte all'interno del codice della libreria: ben 32 volte. Una delle procedure esportate, chiamata `NewRead`, invoca la suddetta funzione ben 11 volte e, in generale, l'invocazione è preceduta quasi sempre da un'altra nei confronti della funzione `ReadRecv`; quest'ultima considerazione dimostra come

quasi certamente gli attaccanti ispezionavano il contenuto dei messaggi ricevuti salvando tutti i dati in **memoria di massa**, affinché fosse accessibile di seguito per un qualche scopo.

rappresenta quella invocata più frequentemente all'interno del codice; ben 32 volte il che dimostra la sua importanza dato il volume di dati registrato dagli attaccanti.

Tale procedura, richiamata molte volte nel codice, ha come scopo quello di scrivere messaggi di log opportunamente formattati in un file esterno, forse per motivi di debug o per tener traccia dello stato di avanzamento dell'attacco. La prima porzione del codice assembly, mostrata nel listato 1.23, è dominata da una grande quantità di istruzioni `std` usate per popolare tutti i registri dalla numero 3 alla 10 e dalla numero 23 alla 31, che probabilmente conterranno i dati da stampare nel file di log. Dal momento che la totalità di queste istruzioni usano i registri `r0` e `r31` come sorgenti quest'ultimi conterranno i dati passati come argomento alla funzione.

Listing 1.23: Codice assembly estratto dal file `2.so`

```
347 mflr     r0
348 std      r23,-72(r1)
349 std      r24,-64(r1)
350 std      r25,-56(r1)
351 std      r26,-48(r1)
352 std      r27,-40(r1)
353 std      r28,-32(r1)
354 std      r29,-24(r1)
355 std      r31,-8(r1)
356 std      r0,16(r1)
357 stdu     r1,-4624(r1)
358 mr       r31,r1
359 std      r4,4680(r31)
360 std      r5,4688(r31)
361 std      r6,4696(r31)
362 std      r7,4704(r31)
363 std      r8,4712(r31)
364 std      r9,4720(r31)
365 std      r10,4728(r31)
366 std      r3,4672(r31)
```

La parte centrale della procedura, mostrata invece nel listato 1.24, contiene un insieme di istruzione il cui scopo evidentemente è quello di scrivere tutti i dati precedentemente raccolti su un file. Come si può facilmente notare dal listato 1.24, è facile intuire che ogni stringhe venga dapprima realizzata facendo ricorso alla funzione standard `snprintf` e poi, dopo l'apertura del file di log attraverso la chiamata di sistema `fopen`, vengano scritti aggiungendo ulteriori informazioni come data e ora locale, come dimostrano le istruzioni di salto verso le procedure `gettimeofday` e `localtime`. La procedura si conclude con una chiamata alla procedura `close` per poi chiudersi definitivamente con l'istruzione `blr` che permette la ritornare alla procedura chiamante.

Listing 1.24: Codice assembly estratto dal file `2.so`

```
347 bl       0x10000748 <.snprintf>
348 ld       r2,40(r1)
```

```
addi     r0,r31,4280
mr       r3,r0
ld       r4,856(r2)
bl       0x10000770 <.fopen>
ld       r2,40(r1)
mr       r0,r3
std      r0,152(r31)
ld       r0,152(r31)
cmpdi    cr7,r0,0
beq      cr7,0x1000a0ec
addi     r0,r31,4264
mr       r3,r0
li       r4,0
bl       0x10000798 <.gettimeofday>
ld       r2,40(r1)
addi     r0,r31,4264
mr       r3,r0
bl       0x100007c0 <.localtime>
```

1.3 Il file 5cfa1c2cb430bec721063e3e2d144feb

fdsfdsfdsfsdfsdf

Tabella 1.7: Dettagli del file 5cfa1c2cb430bec721063e3e2d144feb

Descrizione	**Valore**
Nome	5cfa1c2cb430bec721063e3e2d144feb
Dimensione (*byte*)	1643616
Tipo di file	PE32 executable (GUI) Intel 80386, for MS Windows
MD5 digest	5cfa1c2cb430bec721063e3e2d144feb
SHA1 digest	c1a9044f180dc7d0c87e256c4b9356463f2cb7c6
SHA256 digest	820ca1903a30516263d630c7c08f2b95f7b65dffceb21129c51c9e21cf9551c6
SHA512 digest	a65e615203269b657e55fe842eca0542a4cd3bac80d3039d85dfb5fbbfdb5768 bbabe2fc86f213fb1a759124a82780a1cfbb9fd8457f4923cefad73e9db6f6a4

1.4 Inpatto

Thread Execution Hijacking (a.k.a Suspend, Inject, and Resume (SIR)) This technique has some similarities to the process hollowing technique previously discussed. In thread execution hijacking, malware targets an existing thread of a process and avoids any noisy process or thread creations operations. Therefore, during analysis you will probably see calls to CreateToolhelp32Snapshot and Thread32First followed by OpenThread.

After getting a handle to the target thread, the malware puts the thread into suspended mode by calling SuspendThread to perform its injection. The malware calls VirtualAllocEx and WriteProcessMemory to allocate memory and perform the code injection. The code can contain shellcode, the path to the malicious DLL, and the address of LoadLibrary. Figure 4 illustrates a generic trojan using this technique. In order to hijack the execution of the thread, the malware modifies the EIP register (a register that contains the address of the next instruction) of the targeted thread by calling SetThreadContext. Afterwards, malware resumes the thread to execute the shellcode that it has written to the host process. From the attacker's perspective, the SIR approach can be problematic because suspending and resuming a thread in the middle of a system call can cause the system to crash. To avoid this, a more sophisticated malware would resume and retry later if the EIP register is within the range of NTDLL.dll.

Mitigation This type of attack technique cannot be easily mitigated with preventive controls since it is based on the abuse of operating system design features. For example, mitigating specific Windows API calls will likely have unintended side effects, such as preventing legitimate software (i.e., security products) from operating properly. Efforts should be focused on preventing adversary tools from running earlier in the chain of activity and on identification of subsequent malicious behavior. [77] Identify or block potentially malicious software that may contain process injection functionality by using whitelisting [78] tools, like AppLocker, [79] [80] or Software Restriction Policies [81] where appropriate. [82] Utilize Yama [83] to mitigate ptrace based process injection

by restricting the use of ptrace to privileged users only. Other mitigation controls involve the deployment of security kernel modules that provide advanced access control and process restrictions such as SELinux [84], grsecurity [85], and AppAmour [86].

SELinux (Security Enhanced Linux) è un sistema di controllo degli accessi obbligatorio costruito sull'interfaccia LSM (Linux Security Modules) di Linux. In pratica, il kernel interroga SELinux prima di ogni chiamata di sistema per sapere se il processo è autorizzato ad eseguire una data operazione. SELinux sfrutta una serie di regole, note comunemente come politiche (policy), per autorizzare o vietare operazioni.

AppArmor è un sistema di Mandatory Access Control (Controllo Accesso Obbligatorio) costruito sull'interfaccia LSM (Linux Security Modules) di Linux. In pratica, il kernel interroga AppArmor prima di ogni chiamata di sistema per sapere se il processo è autorizzato ad eseguire una data operazione. Attraverso questo meccanismo, AppArmor confina programmi ad una serie limitata di risorse. AppArmor applica una serie di regole (note come "profilo") su ciascun programma. Il profilo applicato dal kernel dipende dal percorso di installazione del programma in esecuzione. Al contrario di SELinux (discusso nella Sezione 14.5, «Introduzione a SELinux»), le norme applicate non dipendono l'utente. Tutti gli utenti devono sottostare allo stesso insieme di regole quando è in esecuzione lo stesso programma (ma le autorizzazioni utente tradizionali sono ancora valide e potrebbero causare un comportamento diverso!).

Use a firewall to block all incoming connections from the Internet to services that should not be publicly available. By default, you should deny all incoming connections and only allow services you explicitly want to offer to the outside world. Enforce a password policy. Complex passwords make it difficult to crack password files on compromised computers. This helps to prevent or limit damage when a computer is compromised. Ensure that programs and users of the computer use the lowest level of privileges necessary to complete a task. When prompted for a root or UAC password, ensure that the program asking for administration-level access is a legitimate application. Disable AutoPlay to prevent the automatic launching of executable files on network and removable drives, and disconnect the drives when not required. If write access is not required, enable read-only mode if the option is available. Turn off file sharing if not needed. If file sharing is required, use ACLs and password protection to limit access. Disable anonymous access to shared folders. Grant access only to user accounts with strong passwords to folders that must be shared. Turn off and remove unnecessary services. By default, many operating systems install auxiliary services that are not critical. These services are avenues of attack. If they are removed, threats have less avenues of attack. If a threat exploits one or more network services, disable, or block access to, those services until a patch is applied. Always keep your patch levels up-to-date, especially on computers that host public services and are accessible through the firewall, such as HTTP, FTP, mail, and DNS services. Configure your email server to block or remove email that contains file attachments that are commonly used to spread threats, such as .vbs, .bat, .exe, .pif and .scr files. Isolate compromised computers quickly to prevent threats from spreading further. Perform a forensic analysis and restore the computers using trusted media. Train employees not to open attachments unless they are expecting them. Also, do not execute software that is downloaded from the Internet unless it has been scanned for viruses.

Simply visiting a compromised Web site can cause infection if certain browser vulnerabilities are not patched. If Bluetooth is not required for mobile devices, it should be turned off. If you require its use, ensure that the device's visibility is set to "Hidden" so that it cannot be scanned by other Bluetooth devices. If device pairing must be used, ensure that all devices are set to "Unauthorized", requiring authorization for each connection request. Do not accept applications that are unsigned or sent from unknown sources. For further information on the terms used in this document, please refer to the Security Response glossary.

Organizations should ensure that operating systems and all other software are up to date. Software updates will frequently include patches for newly discovered security vulnerabilities that could be exploited by attackers. In all reported FASTCash attacks to date, the attackers have compromised banking application servers running unsupported versions of the AIX operating system, beyond the end of their service pack support dates.

https://www.symantec.com/blogs/threat-intelligence/fastcash-lazarus-atm-malware

What was already known is that the bank robbers inject a malicious Advanced Interactive eXecutive (AIX) executable into a running, legitimate process on the switch application server of an ATM network. These servers are vulnerable and targeted because they are running outdated or unpatched versions of the AIX operating system. The malware is able to create a fraudulent ISO 8583 message, these are the standard for financial transaction messaging. What is now being made public is the fact that the malicious executable is actually malware, named Trojan.Fastcash. The malware has two primary missions: It monitors incoming messages and intercepts attacker-generated fraudulent transaction requests to prevent them from reaching the switch application that processes transactions. It contains logic that generates one of three fraudulent responses to fraudulent transaction requests. The FBI and DHS first issued warnings about Lazarus pulling off ATM attacks, dubbed FASTCash, in early October. Trojan.Fastcash reads all incoming network traffic looking for ISO 8583 messages and the primary account number. "If it finds any containing a PAN number used by the attackers where the Message Type Indicator (MTI) is "0x100 Authorization Request from Acquirer", it will block the message from going any further. It will then transmit a fake response message approving fraudulent withdrawal requests. The result is that attempts to withdraw money via an ATM by the Lazarus attackers will be approved," report stated. Jon DiMaggio, a senior threat intelligence analyst at Symantec, explained this means Lazarus is using a legitimate bank account, usually one with a zero balance. Since the malware intercepts the transaction request for funds, there is no need for the account to have any money at the time of exposure, he said. DiMaggio added there is a likely possibility, although unconfirmed, that Lazarus is using its access to the bank's network create these empty accounts and then use these fraudulent bank account numbers to complete the theft. These newly created accounts and their associated account number are then passed along to the malware so it is ready to act when a withdrawal request comes through. While quite a bit is known about how the attacks take place, there are still some points in the attack that need to be fleshed out. "We have a bit of a blind spot on how the attacker is obtaining the legitimate PANs. However, we believe the attacker is likely using spear phishing emails to obtain initial access to the financial institutions. We also believe the attacker took the time to learn the victim's environment to understand how the financial systems work," he said.

Several versions of Trojan.Fastcash have been spotted each of which uses a different response logic. This could be to tune the malware to be effective against specific banks.

https://www.scmagazine.com/home/security-news/lazarus-fastcash-atm-attack-details-discovered/

Elenco delle figure

Elenco delle tabelle

Listings