

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

Andrea Graziani (0273395)<sup>1</sup>, Alessandro Boccini (0277414)<sup>1</sup>, and  
Ricardo Gamucci (0274716)<sup>1</sup>

<sup>1</sup>Università degli Studi di Roma Tor Vergata

27 marzo 2019

# Indice

<b>1</b>	<b>Analisi tecnica del malware</b>	<b>2</b>
1.1	Il file <code>Injection_API_executable_e</code> . . . . .	3
1.1.1	Analisi delle stringhe . . . . .	3
1.1.2	Analisi del codice assembly . . . . .	10
1.1.2.1	La procedura <code>main</code> . . . . .	10
1.1.2.2	La procedura <code>inject</code> . . . . .	11
1.1.2.3	La procedura <code>proc_attach</code> . . . . .	13
1.2	Il file <code>2.so</code> . . . . .	17
1.2.1	Analisi delle stringhe . . . . .	17
1.2.2	Analisi del codice assembly . . . . .	19
1.2.2.1	La procedura <code>DL_ISO8583_MSG_GetField_Bin</code> .	19
1.2.2.2	La procedura <code>DL_ISO8583_MSG_GetField_Str</code> .	20
1.2.2.3	La procedura <code>DL_ISO8583_MSG_SetField_Bin</code> .	20
1.2.2.4	La procedura <code>DL_ISO8583_MSG_RemoveField</code> .	21
1.2.2.5	La procedura <code>out_dump_log</code> . . . . .	21
1.3	Il file <code>5cfa1c2cb430bec721063e3e2d144feb</code> . . . . .	24

# Capitolo 1

## Analisi tecnica del malware

Secondo il rapporto stilato dalla NCCIC<sup>1</sup>, 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** <sup>2</sup> Usato per l'estrazione di tutte le stringhe stampabili contenuti in un file.

**stat** <sup>3</sup> Utilizzato per ottenere alcune informazioni di base dei file tra cui nome, dimensione, data di ultima modifica, ecc.

**file** <sup>4</sup> Usato per determinare la tipologia di appartenenza di uno specifico file.

**onlinedisassembler** <sup>5</sup> Il de-assemblaggio dei file è stato eseguito utilizzando il servizio cloud **onlinedisassembler** che ci ha permesso di ricavare

---

<sup>1</sup>Cfr. <https://www.us-cert.gov/ncas/analysis-reports/AR18-275A>

<sup>2</sup>Cfr. <https://linux.die.net/man/1/strings>

<sup>3</sup>Cfr. <https://linux.die.net/man/1/stat>

<sup>4</sup>Cfr. <https://linux.die.net/man/1/file>

<sup>5</sup>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...	9ddacbcd0700dc4b9babcd09ac1cebe23a0035099cb612e6c85ff4dff087a26
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<sup>6</sup> fino al 1999<sup>7</sup>, anno della definitiva adozione dello standard **Executable and Linkable Format o ELF**.

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

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

<sup>6</sup>Cfr. <https://it.wikipedia.org/wiki/COFF>

<sup>7</sup>Cfr. [https://en.wikipedia.org/wiki/Executable\\_and\\_Linkable\\_Format](https://en.wikipedia.org/wiki/Executable_and_Linkable_Format)

<sup>8</sup>Cfr. [https://www.ibm.com/support/knowledgecenter/ssw\\_aix\\_72/com.ibm.aix.files/XCOFF.htm](https://www.ibm.com/support/knowledgecenter/ssw_aix_72/com.ibm.aix.files/XCOFF.htm)

<sup>9</sup>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 ( <i>byte</i> )	89088
Data ultima modifica	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<sup>10</sup>, 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<sup>11</sup> e 7.1<sup>12</sup> rispettivamente. Dallo stesso listato si può apprendere inoltre l'architettura del sistema: la PowerPC<sup>TM</sup>.

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)<sup>13</sup>, 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. È 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.<sup>14</sup>

<sup>10</sup>Cfr: <http://www.perzl.org/aix/index.php%3F%3DMain.GCCBinariesVersionNeutral>

<sup>11</sup>Ulteriori dettagli su: <https://gcc.gnu.org/gcc-4.8/>

<sup>12</sup>Ulteriori dettagli su: <https://www-01.ibm.com/support/docview.wss?uid=isg3T1012517>

<sup>13</sup>Cfr: [http://ibmsystemsmag.com/aix/tipstechniques/migration/oslevel\\_versions/](http://ibmsystemsmag.com/aix/tipstechniques/migration/oslevel_versions/)

<sup>14</sup>Cfr: <https://www-01.ibm.com/support/docview.wss?uid=isg3T1012517>

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

```
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<sup>15</sup>, 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 `%11X`<sup>16</sup>.

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=%11X
325 [main] path::%s
326 [main] data(%p)::%s
327 [main] Exec func(%11X) OK
328 [main] Exec func(%11X) fail ret=%X
329 [main] Inject OK(%11X)
330 [main] Inject fail ret=%11X
331 [main] Eject OK
```

<sup>15</sup>Cfr. <https://www.ibm.com/it-it/marketplace/xl-cpp-aix-compiler-power>

<sup>16</sup>Cfr. <http://man7.org/linux/man-pages/man3/printf.3.html>

```

332 [main] Eject fail ret=%llx
333 Usage: injection pid dll_path mode [handle func toc]
334         mode = 0 => Injection
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 suoi 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()`.<sup>17</sup>

Di questi file, alcuni dei quali sono riportati a titolo di esempio nella tabella 1.3<sup>18</sup>, 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.<sup>19</sup>

`/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.<sup>20</sup>

`/proc/pid/status` Contiene informazioni sullo stato del processo.<sup>21</sup>

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.

<sup>17</sup>Cfr. IBM - *AIX Version 7.1: Files References* - pag. 232-246

<sup>18</sup>La lista completa è disponibile in *ivi* pag. 246

<sup>19</sup>Cfr. *ivi* pag. 232

<sup>20</sup>Cfr. *ivi* pag. 232

<sup>21</sup>Cfr. *ivi* pag. 232

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

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

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**<sup>22</sup>, ognuno dei quali identificato da un codice operativo, rappresentato da un valore `int`, e da una serie di argomenti (se presenti)<sup>23</sup>. 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.<sup>24</sup> 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.<sup>25</sup>

**PCRUN** Riesegue un thread dopo essere stato arrestato.

<sup>22</sup>La documentazione IBM usa in modo intercambiabile il termine *messaggio* e quello di *segnale*

<sup>23</sup>Cfr. *ivi* pag. 242

<sup>24</sup>Cfr. *Ibidem*

<sup>25</sup>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.<sup>26</sup>

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<sup>27</sup>

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=%11X
321 [out_regs] MSR=%11X
322 [out_regs] CR=%11X
323 [out_regs] LR=%11X
324 [out_regs] CTR=%11X
325 [out_regs] GPR%d=%11X
```

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 definitivamente con il riavvio del processo attaccato.

<sup>26</sup>Cfr. *ibidem*

<sup>27</sup>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](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](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.

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

```

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
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™

Istruzione	Nome	Argomenti	Descrizione
<b>bl</b>	<i>Branch Link</i>	<i>target_address</i>	<i>Branches to a specified target address.</i>
<b>mfcrr</b>	<i>Move From Condition Register</i>	RT	<i>Copies the contents of the Condition Register into a general-purpose register.</i>
<b>std</b>	<i>STore Doubleword</i>	RS, <i>Offset</i> , RSML	<i>Store a doubleword of data from a general purpose register into a specified memory location.</i>
<b>stw</b>	<i>STore Word</i>	RS, <i>Offset</i> , RSML	<i>Stores a word of data from a general-purpose register into a specified location in memory.</i>
<b>li</b>	<i>Load Immediate</i>	RT, <i>Value</i>	<i>Copies specified value into a general-purpose register.</i>
<b>ld</b>	<i>Load Doubleword</i>	RT, <i>Offset</i> , RS	<i>Load a doubleword of data into the specified general purpose register.</i>
<b>mr</b>	<i>Move Register</i>	RT, RS	<i>Copies the contents of one register into another register.</i>
<b>addi</b>	<i>ADD Immediate</i>	RT, RS, <i>Value</i>	<i>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.</i>
<b>mtrr</b>	<i>Move To Link Register</i>	RS	<i>Copies the contents of RS register into Link Register.</i>
<b>extsw</b>	<i>Extend Sign Word</i>	RT, RS	<i>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).</i>

#### 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 **strtol** (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 inizializzazione, 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 presuppone 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 word 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`.

```

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
11 lwz     r9,0(r9)
12 extsw   r10,r9
13 addi    r9,r31,136
14 mr      r3,r10
15 mr      r4,r9
16 li      r5,8
17 bl      0x100031ec <.write>
18 ld      r2,40(r1)
19 mr      r9,r3
20 stw     r9,128(r31)
21 addi    r9,r2,-768
22 lwz     r9,0(r9)
23 extsw   r29,r9
24 ld      r9,128(r2)
25 lwz     r9,0(r9)
26 extsw   r30,r9
27 ld      r9,128(r2)
28 lwz     r9,0(r9)
29 extsw   r9,r9
30 mr      r3,r9
31 bl      0x1000319c <.strerror>
32 ld      r2,40(r1)
33 mr      r9,r3
34 lwz     r10,128(r31)
35 extsw   r10,r10
36 ld      r3,536(r2)
37 mr      r4,r29
38 mr      r5,r10
39 mr      r6,r30
40 mr      r7,r9
41 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 `mtlr`; 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*).

```

1  ld      r0,16(r1)
2  mtlr    r0
3  ld      r29,-24(r1)
4  ld      r30,-16(r1)
5  ld      r31,-8(r1)
6  blr

1  bl      0x10000674 <.out_log>

```



```

2  bl      0x10001220 <.proc_attach>
3  li      r3,0
4  bl      0x10001a28 <.proc_continue>
5  li      r3,0
6  li      r4,0
7  bl      0x10001b44 <.proc_wait>
8  ld      r3,728(r2)
9  bl      0x10000674 <.out_log>
10 addi    r9,r31,152
11 mr      r3,r9
12 bl      0x10001ee4 <.proc_getregs>
13 addi    r9,r31,152
14 mr      r3,r9
15 bl      0x10000c80 <.out_regs>
16 addi    r8,r31,536
17 addi    r10,r31,152
18 li      r9,384
19 mr      r3,r8
20 mr      r4,r10
21 mr      r5,r9
22 bl      0x1000324c <.memmove>
23 nop
24 ld      r9,536(r31)
25 addi    r9,r9,-16
26 mr      r3,r9
27 li      r4,16384
28 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 finanziarie a proprio favore.

Tabella 1.6: Dettagli del file 2.so

Descrizione	Valore
Nome	2.so
Dimensione ( <i>byte</i> )	110592
Data ultima modifica	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	6890dcce36a87b4bb2d71e177f10ba27f517d1a53ab02500296f9b3aac0218107ced483d70d757a54a5f7489106efa1c1830ef12c93a7f6f240f112c3e90efb5

### 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<sup>28</sup> compatibile con l'architettura PowerPC<sup>TM</sup> con sistema operativo AIX 6.1 (di quest'ultimo il supporto è terminato ufficialmente il 30 Aprile del 2017.<sup>29</sup>).

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
```

<sup>28</sup>Cfr. <http://www.gnu.org/software/gcc/gcc-4.2/>

<sup>29</sup>Cfr. <https://www-01.ibm.com/support/docview.wss?uid=swg21634678#AIX>

```

547 DL_IS08583_MSG_Free
548 DL_IS08583_MSG_SetField_Str
549 DL_IS08583_MSG_SetField_Bin
550 DL_IS08583_MSG_RemoveField
551 DL_IS08583_MSG_HaveField
552 DL_IS08583_MSG_GetField_Str
553 DL_IS08583_MSG_GetField_Bin
554 DL_IS08583_MSG_Pack
555 DL_IS08583_MSG_Unpack
556 DL_IS08583_MSG_Dump
557 DL_IS08583_MSG_AllocField
558 DL_IS08583_COMMON_SetHandler
559 DL_IS08583_DEFS_1987_GetHandler
560 DL_IS08583_DEFS_1993_GetHandler
561 DL_IS08583_FIELD_Pack
562 DL_IS08583_FIELD_Unpack
563 ...

```

---

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

```

545 Blocked Message(msg=%04x, term=%02x, pcode=%06x, pan=%s)
546 Passed Message(msg=%04x, term=%02x, pcode=%06x, pan=%s)
547 [recv] ret=%d
548 send ret = %d, err = %d
549 /tmp/.ICE-unix/context.dat
550 /tmp/.ICE-unix/tmp%d_%d.log
551 [%04d-%02d-%02d %02d:%02d:%02d][PID:%4u][TID:%4u] %s
552 /tmp/.ICE-unix/config_%d
553 /tmp/.ICE-unix/tmp%d_%d.log
554 /tmp/.ICE-unix/tmpwt%d_%d.log
555 [DetourInitFunc] dlopen error(%s)
556 [DetourInitFunc] org_func(%p) %02X %02X %02X %02X %02X %02X
    %02X %02X %02X %02X %02X %02X
557 [DetourInitFunc] new_func(%p) %02X %02X %02X %02X %02X %02X
    %02X %02X %02X %02X %02X %02X
558 [DetourInitFunc] dlsym error(%s)
559 Success
560 Failed
561 DetourInitFunc(%s, %s) %s
562 [DetourInitFunc] org_func=%p new_func=%p
563 [DetourAttach] hook_func_addr=%p, new_func_addr=%p
564 [DetourAttach] after mmap=%p
565 [DetourAttach] copy_func(%x) %02X %02X %02X %02X %02X %02X
    %02X %02X %02X %02X %02X %02X
566 [DetourAttach] hook_func_addr(%x) %02X %02X %02X %02X %02X %02X
    %02X %02X %02X %02X %02X %02X
567 [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 documentazione relativa alla versione R6.8.2 di X11<sup>30</sup> (la famosa implementazione del X Window System<sup>31</sup>) 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*<sup>32</sup>. 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)

```

---

<sup>30</sup>Cfr. <https://www.x.org/releases/X11R6.8.2/doc/RELNOTES5.html>

<sup>31</sup>Cfr. <https://www.x.org/wiki/>

<sup>32</sup>Cfr. [https://www.x.org/releases/X11R7.7/doc/libICE/ICElib.html#Overview\\_of\\_ICE](https://www.x.org/releases/X11R7.7/doc/libICE/ICElib.html#Overview_of_ICE)

```

348  stdu    r1,-80(r1)
349  mr      r31,r1
350  mr      r0,r3
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 interesse 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  clrlldi 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  clrlldi   r9,r0,32
354  lwz      r0,224(r31)
355  clrlldi   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_IS08583_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  clrlldi   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_IS08583_MSG_RemoveField`

La procedura `DL_IS08583_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)
349 addi    r0,r31,4280
350 mr      r3,r0
351 ld      r4,856(r2)
352 bl      0x10000770 <.fopen>
353 ld      r2,40(r1)
354 mr      r0,r3
355 std     r0,152(r31)
356 ld      r0,152(r31)
357 cmpdi   cr7,r0,0
358 beq     cr7,0x1000a0ec
359 addi    r0,r31,4264
360 mr      r3,r0
361 li      r4,0
362 bl      0x10000798 <.gettimeofday>
363 ld      r2,40(r1)
364 addi    r0,r31,4264
365 mr      r3,r0
366 bl      0x100007c0 <.localtime>
```

---



### 1.3 Il file 5cfa1c2cb430bec721063e3e2d144feb

fdsfdsfdsfdsfdf

Tabella 1.7: Dettagli del file 5cfa1c2cb430bec721063e3e2d144feb

Descrizione	Valore
Nome	5cfa1c2cb430bec721063e3e2d144feb
Dimensione ( <i>byte</i> )	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

## Elenco delle figure

# Elenco delle tabelle

1.1	Lista dei file del malware FASTCash . . . . .	3
1.2	Dettagli del file <code>Injection_API_executable_e</code> . . . . .	4
1.3	Sottoinsieme dei file contenuti in <code>/proc/pid</code> . . . . .	7
1.4	Breve descrizione dei registri ispezionati dal malware . . . . .	9
1.5	Alcune istruzioni assembly disponibili nell'architettura PowerPC™	10
1.6	Dettagli del file <code>2.s0</code> . . . . .	17
1.7	Dettagli del file <code>5cfa1c2cb430bec721063e3e2d144feb</code> . . . . .	24

# Listings

1.1	Formato della directory di installazione predefinita delle librerie GCC nei sistemi operativi AIX . . . . .	4
1.2	Stringhe estratte dal file <code>Injection_API_executable_e</code> (1) . . .	4
1.3	Stringhe estratte dal file <code>Injection_API_executable_e</code> (2) . . .	5
1.4	Stringhe estratte dal file <code>Injection_API_executable_e</code> (3) . . .	5
1.5	Stringhe estratte dal file <code>Injection_API_executable_e</code> (4) . . .	7
1.6	Stringhe estratte dal file <code>Injection_API_executable_e</code> (5) . . .	7
1.7	Stringhe estratte dal file <code>Injection_API_executable_e</code> (6) . . .	8
1.8	Stringhe estratte dal file <code>Injection_API_executable_e</code> (7) . . .	8
1.9	Stringhe estratte dal file <code>Injection_API_executable_e</code> (8) . . .	8
1.10	Stringhe estratte dal file <code>Injection_API_executable_e</code> (9) . . .	9
1.11	Codice assembly estratto dal file <code>Injection_API_executable_e</code>	11
1.12	Codice assembly estratto dal file <code>Injection_API_executable_e</code>	11
1.13	Codice assembly estratto dal file <code>Injection_API_executable_e</code>	12
1.14	Codice assembly estratto dal file <code>Injection_API_executable_e</code>	13
1.15	Stringe estratte dal file <code>2.so</code> (1) . . . . .	17
1.16	Stringe estratte dal file <code>2.so</code> (2) . . . . .	17
1.17	Stringhe estratte dal file <code>2.so</code> (3) . . . . .	18
1.18	Stringhe estratte dal file <code>2.so</code> (4) . . . . .	19
1.19	Codice assembly estratto dal file <code>2.so</code> . . . . .	19
1.20	Codice assembly estratto dal file <code>2.so</code> . . . . .	20
1.21	Codice assembly estratto dal file <code>2.so</code> . . . . .	21
1.22	Codice assembly estratto dal file <code>2.so</code> . . . . .	21
1.23	Codice assembly estratto dal file <code>2.so</code> . . . . .	22
1.24	Codice assembly estratto dal file <code>2.so</code> . . . . .	23