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

1.1 Analisi dei file

Tabella 1.1: Lista dei file facentiff parte del malware FASTCash

Nome	**SHA256**
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

1.1.1 Analisi del file `2.so`

In base all'output ottenuto dal tool unix `file`, `2.so` è un file di tipo **eXtended COFF** (**XCOFF**) che rappresenta la versione migliorata ed estesa del formato **Common Object File Format** (**COFF**), il formato di file standard che ha definito la struttura dei file eseguibili e delle librerie nei sistemi operativi UNIX[1] fino al 1999[2], anno della definitiva adozione dello standard **Executable and Linkable Format** o **ELF**. XCOFF rappresenta tuttavia uno standard proprietario sviluppato da IBM[3] adottato nei sistemi operativi **Advanced Interactive eXecutive** o **AIX**, una famiglia di sistemi operativi proprietari basati su Unix sviluppati dalla stessa IBM.[4]

In accordo alle nostre analisi, confermate anche dal report AR18-275A della NCCIC, il file `file 2.so` rappresenta una **shared library** che esporta una serie di metodi che consentono l'iterazione con i sistemi finanziari che utilizzano il protocollo **ISO8583**.[5]

Tabella 1.2: Dettagli del file `2.s0`

Descrizione	Valore	Comando Unix
Nome	`2.so`	`stat -c "%n" 2.so`
Dimensione (*byte*)	`110592`	`stat -c "%s" 2.so`
Data ultima modifca	`2018-11-09 11:08:40.000000000 +0100`	`stat -c "%y" 2.so`
Tipo di file	`64-bit XCOFF executable or object module`	`file 2.so`
MD5	`b66be2f7c046205b01453951c161e6cc`	`md5sum 2.so`
SHA1	`ec5784548ffb33055d224c184ab2393f47566c7a`	`sha1sum 2.so`
SHA256	`ca9ab48d293cc84092e8db8f0ca99cb1 55b30c61d32a1da7cd3687de454fe86c`	`sha256sum 2.so`
SHA512	`6890dcce36a87b4bb2d71e177f10ba27f517d1a53ab02500296f9b3aac021810 7ced483d70d757a54a5f7489106efa1c1830ef12c93a7f6f240f112c3e90efb5`	`sha512sum 2.so`

1.1.1.1 Stringhe stampabili rilevanti

Per estrazione di tutte le stringhe stampabili contenute nel file `2.so` ci siamo serviti del tool `strings`[6] di cui riportiamo frammenti dell'output ottenuto nei listati **??** e **??**.

Listing 1.1: Stringe estratte dal file `2.so`

```
...
_GLOBAL__FI_eg64_so
_GLOBAL__FD_eg64_so
=s4m
```

[1] Cfr. `https://it.wikipedia.org/wiki/COFF`

[2] Cfr. `https://en.wikipedia.org/wiki/Executable_and_Linkable_Format`

[3] Cfr. IBM - *XCOFF Object File Format* - `https://www.ibm.com/support/knowledgecenter/ssw_aix_72/com.ibm.aix.files` (data ultima consultazione 27-03-2019)

[4] Cfr. `https://www.ibm.com/it-infrastructure/power/os/aix`

[5] Cfr. The National Cybersecurity and Communications Integration Center's (NCCIC), *Malware Analysis Report (AR18-275A)* - 2 Ottobre 2018 - `https://www.us-cert.gov/ncas/analysis-reports/AR18-275A)`

[6] Cfr. `https://linux.die.net/man/1/strings`

```
469 /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
470 libc.a
471 shr_64.o
472 libpthread.a
473 shr_xpg5_64.o
474 ...
```

Poiché nei sistemi operativi AIX la directory all'interno del quale sono contenute tutte le librerie di GCC assume la forma mostrata nel listato **??**[7], possiamo dedurre dalla riga 496 del listato **??** che la versione di GCC utilizzata è stata la 4.2.0 (versione rilasciata il 13 Maggio 2007[8]) mentre la versione del sistema operativo bersaglio fosse stata la V6.1, versione ormai obsoleta del sistema operativo AIX il cui supporto è terminato ufficialmente il 30 Aprile del 2017.[9] Dalla stessa riga osserviamo che l'architettura hardware del sistema bersaglio è equipaggiata con un processore PowerPC

Ovviamente il riferimento alla libreria standard `libc.c` e di GCC suggeriscono che il malware è stato scritto in C/C++.

Listing 1.2: Formato del percorso di installazione delle librerie GCC nei sistemi operativi AIX

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

Il listato **??** mostra ciò che dovrebbero essere i nomi delle procedure esportate dalla libreria il che dimostra in modo inequivocabile il fatto che il malware è in grado di interagire con i sistemi informatici che fanno uso del protocollo ISO8583.

Listing 1.3: Stringhe estratte dal file `2.so`

```
545 ...
546 DL_ISO8583_MSG_Init
547 DL_ISO8583_MSG_Free
548 DL_ISO8583_MSG_SetField_Str
549 DL_ISO8583_MSG_SetField_Bin
550 DL_ISO8583_MSG_RemoveField
551 DL_ISO8583_MSG_HaveField
552 DL_ISO8583_MSG_GetField_Str
553 DL_ISO8583_MSG_GetField_Bin
554 DL_ISO8583_MSG_Pack
555 DL_ISO8583_MSG_Unpack
556 DL_ISO8583_MSG_Dump
557 _DL_ISO8583_MSG_AllocField
558 DL_ISO8583_COMMON_SetHandler
559  DL_ISO8583_DEFS_1987_GetHandler
560  DL_ISO8583_DEFS_1993_GetHandler
```

[7] http://www.perzl.org/aix/index.php%3Fn%3DMain.GCCBinariesVersionNeutral
[8] http://www.gnu.org/software/gcc/gcc-4.2/
[9] https://www-01.ibm.com/support/docview.wss?uid=swg21634678#AIX

```
561 _DL_ISO8583_FIELD_Pack
562 _DL_ISO8583_FIELD_Unpack
563 ...
```

1.1.1.2 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 1.3, 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.

1.1.1.2.1 La procedura `DL_ISO8583_MSG_GetField_Bin` Analizzando le prime righe di codice della procedura, riportate in parte nel listato 1.4, 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 1.4: Stringhe estratte dal file `Injection_API_executable_e`

```
347 std     r31,-8(r1)
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 1.5, 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 1.5: Stringhe estratte dal file `Injection_API_executable_e`

```
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
```

1.1.1.2.2 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.

1.1.1.2.3 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 1.5, 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 1.6: Stringhe estratte dal file `Injection_API_executable_e`

```
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>
```

1.1.1.2.4 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 1.7, riguarda l'eliminazione del campo specificato che avviene attraverso una chiamata alla procedura `free`.

Listing 1.7: Stringhe estratte dal file `Injection_API_executable_e`

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

1.1.1.2.5 La procedura `out_dump_log`

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 1.8, 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 1.8: Stringhe estratte dal file `Injection_API_executable_e`

```
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 1.9, 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 1.9, 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 1.9: Stringhe estratte dal file `Injection_API_executable_e`

```
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>
```

1.1.2 Analisi del file `Injection_API_executable_e`

In questa sezione dimostreremo come il file di tipo **eXtended COFF** denominato `Injection_API_executable_e` sia in grado di eseguire un attacco di **code injection** a danno di un processo in esecuzione in modo tale da modificarne il comportamento a favore degli attaccanti.

Tabella 1.3: Dettagli tecnici del file `2.s0`

Descrizione	Valore	Comando Unix
Nome	`2.so`	`stat -c "%n" 2.so`
Dimensione (*byte*)	`89088`	`stat -c "%s" 2.so`
Data ultima modifca	`2018-11-09 11:08:40.000000000 +0100`	`stat -c "%y" 2.so`
Tipo di file	`64-bit XCOFF executable or object module`	`file 2.so`
MD5	`b3efec620885e6cf5b60f72e66d908a9`	`md5sum 2.so`
SHA1	`274b0bccb1bfc2731d86782de7babdeece379cf4`	`sha1sum 2.so`
SHA256	`d465637518024262c063f4a82d799a4e 40ff3381014972f24ea18bc23c3b27ee`	`sha256sum 2.so`
SHA512	`a36dab1a1bc194b8acc220b23a6e36438d43fc7ac06840daa3d010fddcd9c316 8a6bf314ee13b58163967ab97a91224bfc6ba482466a9515de537d5d1fa6c5f9`	`sha512sum 2.so`

1.1.2.1 Stringhe stampabili rilevanti

Cominciamo lo studio del file `Injection_API_executable_e` partendo dall'analisi delle stringhe stampabili estratte attraverso il tool `strings`. Seguendo lo stesso ragionamento precedentemente descritto nella sezione 1.1.1.1, possiamo osservare dal listato 1.10 la versione di GCC e del sistema operativo AIX utilizzati per eseguire la *build* del malware, che risultano essere rispettivamente pari a 4.8.5 (la data pubblicazione risale al 23 giugno 2015[10]), e 7.1 (commercializzata a partire da settembre 2010).[11] Sfortunatamente non è stato possibile risalire alla versione degli aggiornamenti, che IBM identifica con il nome di *Technology Levels* (TLs) [12], installati sul sistema operativo bersaglio al momento dell'attacco in modo tale da conoscere l'entità del rischio a cui si sottoponeva il sistema bancario. In ogni caso il supporto ufficiale per la versione 7.1, sostituita dalla ben più moderna versione 7.2 rilasciato nel dicembre 2015, è già terminato il 30 Novembre 2013 benché la versione 7.1 TL5 riceverà supporto fino ad aprile 2022.[13]

Listing 1.10: Stringhe estratte dal file `Injection_API_executable_e`

```
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 ...
```

[10] Cfr. https://gcc.gnu.org/gcc-4.8/
[11] Cfr: https://www-01.ibm.com/support/docview.wss?uid=isg3T1012517
[12] http://ibmsystemsmag.com/aix/tipstechniques/migration/oslevel_versions/
[13] Cfr: https://www-01.ibm.com/support/docview.wss?uid=isg3T1012517

Un altro riferimento ai tool utilizzati dagli attaccanti lo possiamo ricavare dalla riga 944 riportata nel listato 1.11 dove apprendiamo l'utilizzo del compilatore lo **XL C/C++ for AIX** versione 11.1.0.1, quest'ultimo appositamente ottimizzato dalla IBM per i propri sistemi operativi. Ci risulta che tale versione del compilatore non fosse disponibile per AIX 7.1 al momento del rilascio e che sia divenuto disponibile in seguito ad un aggiornamento. [14]

Listing 1.11: Stringhe estratte dal file `Injection_API_executable_e`

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

Analizziamo ora in dettaglio le varie operazioni compiute dal malware durante la sua esecuzione. Innanzitutto, osservando la particolare configurazione dei listati successivi come la numero 1.13, notiamo quello che dovrebbe essere un insieme di stampe nella forma `[FUNCTION NAME] [...]` eseguito probabilmente da un meccanismo di log, il che è stato confermato dalla già citata analisi della NCCIC; infatti, notiamo un gran numero di stringhe contenenti i ben noti *conversion specifier* utilizzati nelle stringhe che specificano il formato delle stampe eseguite dalla funzione `fprintf` di cui molti sono nella forma %llX, usata per stampare numeri interi senza segno in forma esadecimale[15]. Come è stato confermato da altre analisi, le stringhe riportate nelle righe 333, 334 e 335 suggeriscono come l'applicazione è stata progettata per essere una command-line utility interattiva e di come il meccanismo di log sia stato utilizzato per ottenere informazioni e consentire agli attaccanti un attacco mirato.

Listing 1.12: Stringhe estratte dal file `Injection_API_executable_e`

```
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
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
```

[14] https://www-01.ibm.com/support/docview.wss?uid=swg21326972
[15] Cfr. http://man7.org/linux/man-pages/man3/printf.3.html

```
341 [main] WARNING::func_addr(%llX), toc_addr(%llX)
342 ...
```

La presenza delle stringhe `out_log`, `file_dump` suggeriscono che l'output dei log venisse redirezionato verso file esterni. Forse è previsto un meccaniscmo di caricamento delle impostazioni.

Listing 1.13: Stringhe estratte dal file `Injection_API_executable_e`

```
320 x}8KxH
321 out_log
322 }CSx8
323 ...
324  }*J
325 store_config
326 }CSx8
327 ...
328 }#Kx8?
329 load_config
330 }CSx8
331 ...
332  }*J
333 file_dump
334 @}$KxK
```

Prima di descrivere le varie fasi dell'attacco, è indispensabile comprendere come vengono rappresentati e gestiti i **processi** nei sistemi operativi AIX versione 6.1 e 7.1. Ogni processo del sistema viene opportunamente rappresentato da un insieme di file ognuno dei descrive un particolare aspetto di un processo come ad esempio il suo stato, le informazioni sui file descriptors, privilegi ecc. Tutti i processi sono raccolti all'interno della directory `/proc` mentre tutti i file di un dato processo con identificatore pari a `pid` sono raccolti all'interno della directory `/proc/pid`; ciò permette agli attaccanti di ricavare tutte le informazioni necessarie di tutto il sistema attraverso le system call standard per la lettura e scrittura sui file come `open()`, `close()`, `read()` e la `write()`. [16] Di tutti i file contenuti in una generica directory `/proc/pid`, di cui ne riportiamo una piccola frazione nella tabella 1.4[17], è importante ricordare:

/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.[18]

/proc/pid/ctl Un file di sola scrittura in cui vengono scritti messaggi strutturati che consentono di modificare lo stato del processo e dunque il suo comportamento. I messaggi di controllo vengono scritti direttamente sul file ctl del processo e gli effetti sono visibili immediatamente attraverso i file di stato del processo.[19]

[16] Cfr. IBM - *AIX Version 7.1: Files References* - pag. 232-246
[17] Per una lista completa *Cfr. ivi* pag. 246
[18] *Cfr. ivi* pag. 232
[19] *Cfr. ivi* pag. 232

/proc/pid/status Contiene informazioni sullo stato del processo. [20]

Tabella 1.4: 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`

Dal momento che, come mostrato nel listato 1.14, sono stati individuate tre stringhe che fanno riferimento ai suddetti file descrittori di processo, in particolare ai file `ctl`, `status` e `as`, si può affermare che il malware, ricostruendo probabilmente le directory attraverso una chiamata `sprintf` come dimostrano la presenza del *conversion specifier* `%d` e la presenza di varie stringhe `sprintf` (riga 372 e 824), è stato progettato per accedere a questi file con lo scopo di manipolare il normale flusso di esecuzione del processo bersaglio.

Listing 1.14: Stringhe estratte dal file `Injection_API_executable_e`

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

Come abbiamo detto in precedenza, il flusso di esecuzione di un processo può essere modificato eseguendo la scrittura di appositi messaggi nel file `ctl` i quali da un codice operativo rappresentato da un int che identifica la specifica operazione seguita da ulteriori argomenti (se presenti).[21] Il listato 1.16 dimostra che il malware interrompe esplicitamente l'esecuzione del processo bersaglio attraverso l'uso del messaggio **PCWSTOP** il cui scopo è quello di arrestare l'esecuzione di un processo `pid` passato come argomento.[22]

Listing 1.15: Stringhe estratte dal file `Injection_API_executable_e`

```
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 ...
```

Il listato 1.16 mostra invece l'uso di vari messaggi tra cui:

PCSET Serve per impostare una serie di flag ad un processo (PR_ASYNC, PR_FORK, PR_KLC ecc.).[23]

[20] *Cfr. ivi* pag. 232
[21] *Cfr. ivi* pag. 242
[22] *Ibidem*
[23] *Cfr. ivi* pag. 234

PCRUN Riesegue un thread dopo essere stato arrestato.

PCSENTRY Indica al thread di interrompere la sua esecuzione 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.

Listing 1.16: Stringhe estratte dal file `Injection_API_executable_e`

```
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 ...
```

I listati 1.17 e 1.18 dimostrano come il malware raccolga le informazioni necessarie al suo scopo attraverso l'accesso in lettura alle informazioni di stato del processo e al contenuto dei registri. Abbiamo riportato nella tabella 1.5 una descrizione dei registri ispezionati dal malware[24]. del processore tra cui risultano

Listing 1.17: Stringhe estratte dal file `Injection_API_executable_e`

```
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 1.18: Stringhe estratte dal file `Injection_API_executable_e`

```
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
```

https://www.ibm.com/support/knowledgecenter/en/ssw_aix_71/com.ibm.aix.kdb/kdb_registers.htm

Infine nei listati 1.20 e 1.19 viene mostrato l'accesso in lettura e in scrittura allo spazio di indirizzamento del processo bersaglio e di come riavvi l'esecuzione del processo dopo la code injection.

[24]Cfr. IBM - *AIX Version 7.1: Assembler Language Reference*

Tabella 1.5: 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 1.19: Stringhe estratte dal file `Injection_API_executable_e`

```
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 ...
```

Listing 1.20: Stringhe estratte dal file `Injection_API_executable_e`

```
308 ...
309 [proc_continue] PCRUN pid=%d, arg=%d, ret=%d, err=%d(%s)
310 ...
```

PowerPC 601 RISC Microprocessor EA (effective address)

1.1.2.2 Disassemblaggio

Tabella 1.6: Alcune istruzioni assembly disponibili nell'architettura PowerPC™

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).*
`cmpdi`	*CoMPare Doubleword Immediate*	`RS,` *`Value`*	

1.1.2.2.1 Analisi della 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 1.21 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 1.21: Stringhe estratte 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>
```

1.1.2.2.2 Analisi della 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 1.22 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 1.22: Stringhe estratte 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 1.23 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 1.23: Stringhe estratte 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 1.24, 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 1.24: Stringhe estratte 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>
```

1.1.2.2.3 Analisi della 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 <.proc_attach>:
```

```
mflr    r0
std     r0,16(r1)
std     r29,-24(r1)
std     r30,-16(r1)
std     r31,-8(r1)
stdu    r1,-352(r1)
mr      r31,r1
li      r9,0
stw     r9,128(r31)
li      r9,0
stw     r9,132(r31)
li      r9,0
std     r9,136(r31)
li      r9,0
std     r9,144(r31)
li      r9,0
std     r9,152(r31)
li      r9,0
std     r9,160(r31)
li      r9,0
std     r9,168(r31)
li      r9,0
stw     r9,176(r31)
addi    r10,r31,180
li      r9,140
mr      r3,r10
li      r4,0
mr      r5,r9
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`.

```
li      r9,14
stw     r9,136(r31)
li      r9,4
stw     r9,140(r31)
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>
```

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

```
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>
```

La traduzione dal linguaggio macchina all'assembler del file è stato usufrendo del servizio web https://onlinedisassembler.com/ per motivi di semplicità con le seguenti impostazioni

architettura powerpc620 processore POWER 7 64 bit

Queste impostazioni ci hanno permesso di ottenere un output sostanzialmente identico a quello mostrato in vari screenshot dalla CISA

ftp://public.dhe.ibm.com/systems/power/docs/aix/72/idalangref_pdf.pdf

Specifies a 24-bit signed two's-complement integer that is concatenated on the right with 0b00 and sign-extended to 64 bits (PowerPC®) or 32 bits (POWER® family). This is an immediate field.

https://www.ibm.com/support/knowledgecenter/ssw_aix_72/com.ibm.aix.alangref/idalangref_inst_fiel

mflr r0 # move LR into GPR0

If a branch instruction has the Link bit set to 1, then the Link Register is altered to store thereturn address for use by an invoked subroutine. The return address is the address of the instructionimmediately following the branch instruction (pag 33)

The following code transfers the execution of the program to here and sets the Link Register:

https://www.ibm.com/support/knowledgecenter/en/ssw_aix_71/com.ibm.aix.alangref/idalangref_bbra

Elenco delle figure

Elenco delle tabelle

Listings