Analisi tecnica e considerazioni sul malware Strela

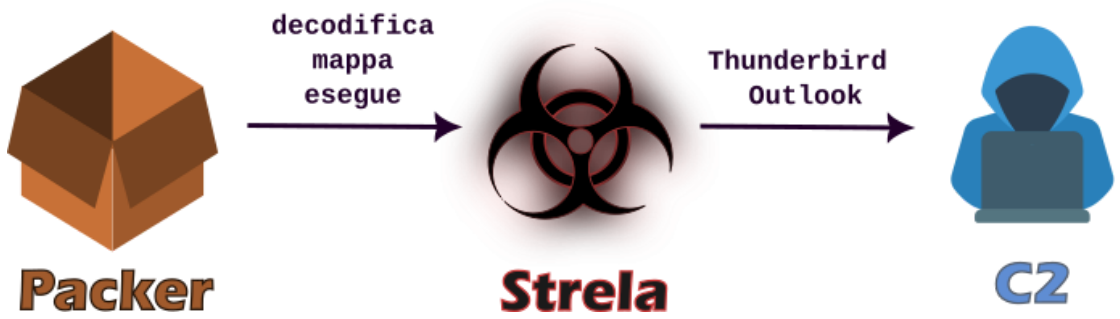
La scorsa settimana il malware Strela è approdato in Italia. **Strela** è un semplice *stealer* specializzato nel furto delle credenziali di posta dagli applicativi **Thunderbird** e **Outlook**. Il malware in sè è piuttosto semplice ma, fatta eccezione per la prima ondata, il packer con cui viene veicolato è più complicato da analizzare per via della della tecnica di Control Flow Obfuscation (CFO) che impiega.

Vogliamo in questo articolo analizzare il malware, dividendo le analisi in tre sezioni:

1. nella prima (e più tecnica) parte discuteremo alcune tecniche per analizzare il packer di Strela e scriveremo uno script per l'estrazione automatica del payload (che si rileverà estremamente semplice a discapito della complessità iniziale);
2. nella seconda, descriveremo Strela stesso incluso il metodo utilizzato per inviare i dati al C2;
3. nella terza parte caratterizzeremo proprio il C2.

Per la sola descrizione malware Strela è possibile passare subito alla relativa sezione.

L'infezione non usa TTP peculiari ed è essenzialmente un classico esempio di malware diffuso tramite packer. Una volta che il payload è in esecuzione, questo ruba gli account Thunderbird ed Outlook e li invia al C2.

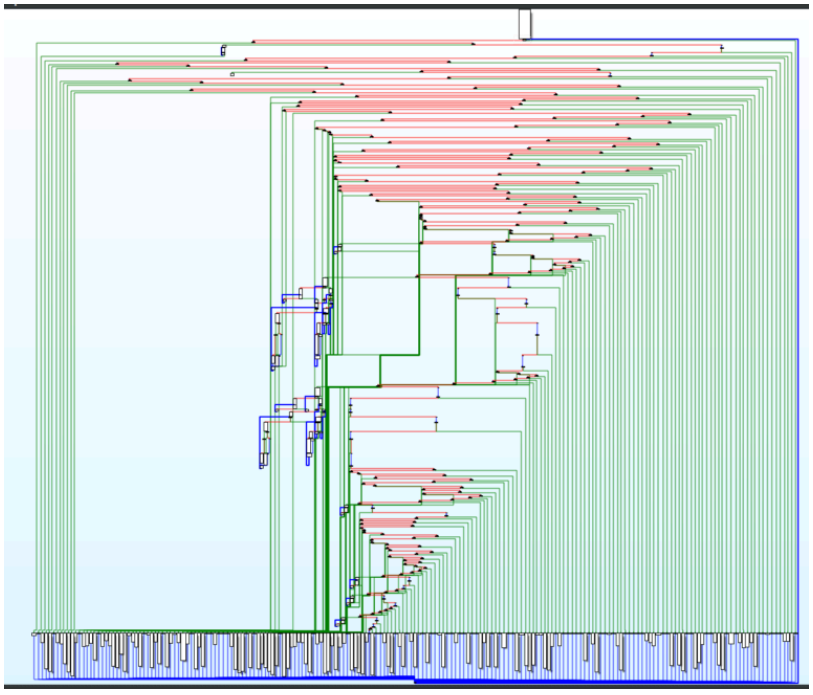


Le fasi dell'infezione di Strela

Il packer di Strela

Il packer sembra scritto con MinGW (di cui si riconoscono molte funzioni firma) ed il debugger IDA non ha particolari problemi ad identificare buona parte del runtime. Sono presenti due callback TLS ma sono quelli tipici di MinGW e per i quali non è necessaria che una veloce analisi superficiale.

La funzione `WinMain` è però stata offuscata facendo uso, come già anticipato, di CFO. Il CFG generato da IDA è decisamente disarmante.



Il grafico di esecuzione della principale funzione del packer (WinMain)

La procedura di analisi più diretta (ma più complessà) da seguie è quella che cerca di districare il flusso di esecuzione in modo simile a come era stato fatto per Emotet a suo tempo. Prima di vedere come affrontare un simile problema è utile delineare brevemente alcune tecniche più immediate per il recupero del payload.

Analisi dinamica

Navigando sommariamente i blocchi della funzione WinMain si può notare come molti di essi siano composti da istruzioni aritmetiche il cui unico scopo è quello di complicare la comprensione del codice. Dato che l'unica sezione eseguibile è quella del codice e che questa è in sola lettura, un eventuale payload o shellcode deve essere decodificato in un'area appositamente allocata con i permessi di scrittura ed esecuzione. Torna utile quindi cercare istruzioni di chiamate o salti indiretti: le prime possono essere indicatrici di chiamate ad API, i secondi dell'inizio dell'esecuzione del payload.

Sebbene sia possibile effettuare ricerce più specifiche con IDA, una ricerca testuale della stringa "*call*" è sufficiente per trovare una chiamata a `VirtualAlloc`.

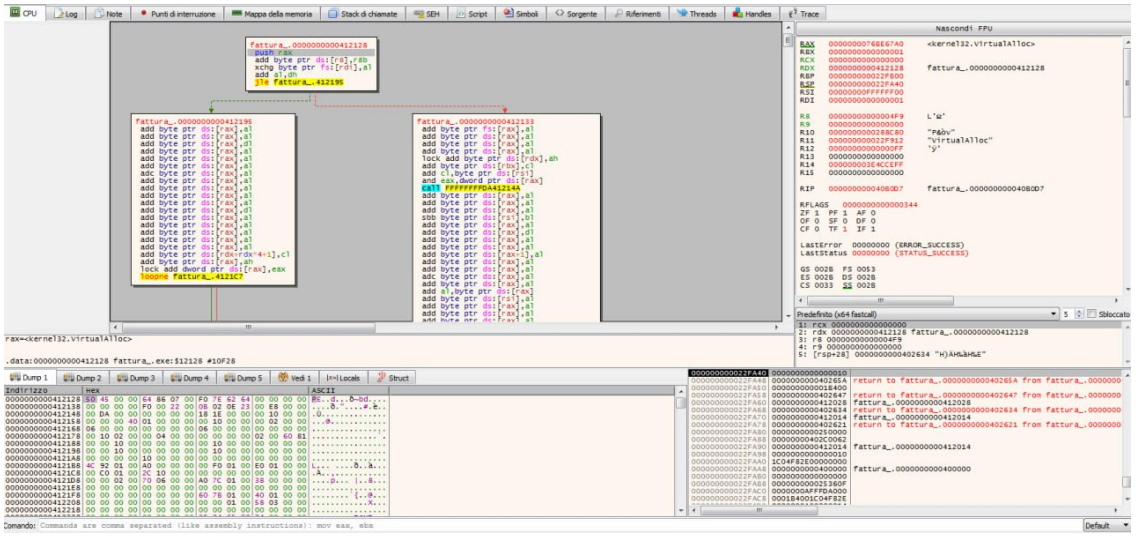
```
mov     rax, [rbp+350h+var_350]
mov     rax, [rax]
mov     [rbp+350h+var_1C0], rax
mov     rax, [rbp+350h+var_1C0]
mov     [rbp+350h+var_1C8], rax
mov     rax, [rbp+350h+var_1C0]
mov     rcx, [rbp+350h+var_1C8]
movsxd  rcx, dword ptr [rcx+3Ch]
add     rax, rcx
mov     [rbp+350h+var_1D0], rax
mov     rax, qword ptr cs:aVirtualalloc ; "VirtualAlloc"
mov     qword ptr [rbp+350h+ProcName], rax
mov     edx, dword ptr cs:aVirtualalloc+8 ; "lloc"
mov     [rbp+350h+var_1D5], edx
mov     r8b, byte ptr cs:aVirtualalloc+0Ch ; ""
mov     [rbp+350h+var_1D1], r8b
sub     rsp, 20h
lea     rcx, LibFileName ; "Kernel32"
call    cs:LoadLibraryA
add     rsp, 20h
lea     rdx, [rbp+350h+ProcName] ; lpProcName
sub     rsp, 20h
mov     rcx, rax        ; hModule
call    cs:GetProcAddress
add     rsp, 20h
xor     r9d, r9d
mov     ecx, r9d
mov     rdx, [rbp+350h+var_1D0]
mov     r9d, [rdx+50h]
mov     edx, r9d
sub     rsp, 20h
mov     r8d, 3000h
mov     r9d, 40h ; '@'
call    rax
add     rsp, 20h
mov     [rbp+350h+var_1E8], rax
mov     rax, [rbp+350h+var_1D0]
add     rax, 108h
mov     [rbp+350h+var_1F0], rax
mov     [rbp+350h+var_1F4], 0
```

Il blocco chiave per l’analisi dinamica del packer

Sebbene il codice sia ordinario ed è facile sorvolarci sopra, ci sono due dettagli importanti da notare:

1. I permessi delle pagine allocate sono PAGE_EXECUTE_READWRITE. Questo indica che abbiamo trovato il buffer ipotizzato precedentemente.
2. La dimensione del buffer allocato (che, in quanto secondo parametro, si troverà in `RDX` prima della chiamata) è ottenuta leggendo da `[rdx+50h]`, dove `RDX` è la base di una struttura dati. 0x50 è l’offset del campo ImageSize nell’*Optional Header PE*: possiamo quindi ipotizzare che il packer stia allocando l’area di memoria in cui mappare il PE.

Trattandosi di un’analisi dinamica, non ci rimane che piazzare un breakpoint dopo la chiamata a `GetProcAddress` e verificare se l’ipotesi è corretta. Dopo una manciata di Single Step otteniamo il PE di Strela:



L’header PE del payload (Strela) decodificato.

Sempre mantenendoci nell’ambito dell’analisi dinamica, altri possibili approcci sono:

1. mettere un breakpoint sull’intero buffer allocato e, una volta individuato il codice che vi scrive, identificare da dove sono letti i dati. Questo fornità la posizione del payload;
2. cercare in memoria un header PE. E’ necessario essere consapevoli delle limitazioni di questo secondo metodo: ad esempio il packer decodifca l’intero PE di Strela ma poi non ne mappa l’header e sono ben noti casi in cui l’header PE è volutamente alterato per evitare match per firma.

Questo è sicuramente il metodo più veloce da usare quando ci si trova di fronte alla prima analisi di un nuovo malware ed è il metodo che abbiamo usato per rispondere velocemente alla campagna ed estrarre e censire i relativi IoC.

Crittonalisi

Un altro metodo di analizzare il packer è quello di cercare di capire come e dove è salvato il payload codificato. Generalmente questa è una battaglia generalmentepersa in partenza se non si ha a disposizione il codice per guidarci poiché vi è un’infinità numerabile di codec utilizzabili e dedurre l’algoritmo usato dai soli dati è quasi sempre impossibile.

Tuttavia, vale la pena menzionare questo metodo perchè:

1) funziona in questo caso specifico;

2) è una utile forma mentis.

Di tutte le sezioni PE presenti nel packer, quella `.data` è la più grande. Anche da IDA è possibile capire che è i dati sono più numerosi del codice stesso.

La sezione dati (verde e grigio) è più grande di quella del codice.

Questo ci fa pensare che il payload sia effettivamente nella sezione `.data`. Sebbene possa sembrare scontato che sia così (il payload è composto da dati) va ricordato che in realtà, trattandosi di dati in sola lettura, questi possono essere presenti ovunque (es: nelle risorse o nella sezione del codice stesso).

E' ben noto che i dati cifrati a dovere non sono (polinomialmente) distinguibili da dati pseudo-casuali quindi dobbiamo cercare necessariamente dei pattern.

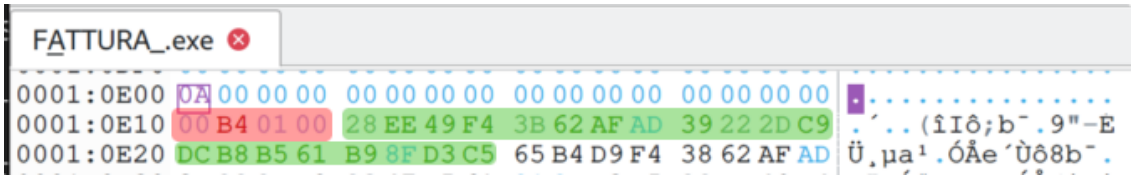
L'immagine sotto mostra la sezione dati del packer.

La sezione dati del packer, si notano alcuni pattern.

Sebbene inizialmente sia difficile notarli, sono presenti dei pattern. A parte il byte di valore `0x0a` (che una veloce ricerca su IDA mostra essere usato dal runtime di *MinGW*), il primo particolare che salta all'occhio è la DWORD di valore `0x1b400` (che è presente in little endian quando visualizzata a byte). Vista la cifra tonda, questa probabilmente rappresenta la dimensione del payload. Il valore è inoltre molto simile alla dimensione della sezione di dati stessa, rafforzando questa ipotesi.

I dati seguenti però non sembrano rilevare altri pattern. Tuttavia, un'analisi più attenta mostra la ripetizione di certi byte. Ad esempio i caratteri ripetuti che più saltano all'occhio sono `ÉÜ`, ripetuti ogni 20 byte. Questo potrebbe essere sintomatico dell'utilizzo di un rolling XOR per la codifica del payload. Dato che un PE ha naturalmente molte sequenze di zeri, queste espongono in chiaro la chiave in caso di cifratura con XOR.

Ipotizzando un rolling XOR, la chiave potrebbe essere lunga 20 byte ma non sappiamo quale sia l'inizio. Ispezionare l'inizio dei dati può aiutarci a capirlo. Dato che i caratteri `ÉÜ` compaiono poco dopo l'inizio dei dati, è probabile che la chiave sia in chiaro subito dopo la dimensione del payload. Possiamo quindi ipotizzare il formato `<dimensione><chiave xor><payload>` .



In rosso la dimensione del payload. In verde la chiave XOR di decifratura. A seguire il payload cifrato.

Una rapida verifica su **Cyberchef** ci conferma che le nostre ipotesi erano esatte. E' ora possibile scrivere un semplice script Python per l'estrazione automatica di Strela:

```
#!/usr/bin/env python3 import struct, sys def read_pe_section(filename: str, section_name: str) -> bytearray: pe = None sections = {} image_base = None def read_u64(x: int) -> int: return struct.unpack("<Q", pe[x:x+8])[0] def read_u32(x: int) -> int: return struct.unpack("<I", pe[x:x+4])[0] def read_u16(x: int) -> int: return struct.unpack("<H", pe[x:x+2])[0] def at_va(va: int, is_rva : bool = False) -> slice: rva = va - (image_base if not is_rva else 0) for s in sections.values(): if rva >= s["rva"] and rva < s["rva"] + s["va_size"]: return slice(rva - s["rva"] + s["raw_offset"], s["raw_size"] + s["raw_offset"]) raise ValueError("RVA is out of range") with open(filename, "rb") as f: pe = f.read() nt_header_offset = read_u32(0x3c) n_sections = read_u16(nt_header_offset + 0x6) n_sections_offset = nt_header_offset + read_u16(nt_header_offset + 0x14) + 0x18 image_base = read_u64(nt_header_offset + 0x30) sections = {} for i in range(n_sections): section_start_offset = n_sections_offset + 0x28 * i name = pe[section_start_offset : section_start_offset + 0x8].rstrip(b"\x00").decode("iso-8859-1") va_size = read_u32(section_start_offset + 0x8 ) rva = read_u32(section_start_offset + 0x0c ) raw_size = read_u32(section_start_offset + 0x10 ) raw_offset = read_u32(section_start_offset + 0x14 ) sections[name] = dict(rva=rva, va_size=va_size, raw_size=raw_size, raw_offset=raw_offset) return bytearray(pe[at_va(sections[section_name]["rva"], is_rva = True)]) def decode_payload(data_section: bytearray, key_len: int = 0x14, struct_offset:int = 0x10) -> bytes: payload_size = struct.unpack("<I", data_section[struct_offset:struct_offset+4])[0] payload_key = data_section[struct_offset+4:struct_offset+4+key_len] payload = data_section[struct_offset+4+key_len:struct_offset+4+key_len + payload_size] for i in range(len(payload)): payload[i] ^= payload_key[i % len(payload_key)] return payload def main(): if len(sys.argv) != 3: print("Usage: ex.py INPUT OUTPUT", file=sys.stderr) sys.exit(1) with open(sys.argv[2], "wb") as f: f.write(decode_payload(read_pe_section(sys.argv[1], ".data"))) if __name__ == "__main__": main()
```

In questo caso siamo stati fortunati poichè il packer utilizza una codifica semplice e riconoscibile ma, in generale, senza il codice a fare da guida non è possibile determinare come estrarre automaticamente il payload.

Anche se l'esperienza ci ha insegnato che non è conveniente spendere troppo tempo nel cercare di estrarre automaticamente un malware (vista l'estrema variabilità dei packer), in questo specifico caso abbiamo utilizzato

quanto appreso per velocizzare notevolmente l'estrazione degli IoC dalle campagne Strela che si sono verificate la scorsa settimana e che sono riprese anche in data odierna. Come vedremo, Strela non ha offuscazione e questo script ci ha permesso di estrarre gli IoC senza la necessità di aprire una VM per l'analisi.

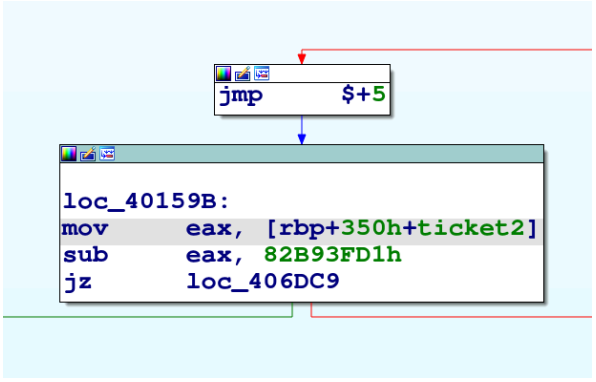
Analisi statica

A questo punto sappiamo più o meno tutto quello che ci interessava sapere sul packer ma, paradossalmente, non sappiamo niente del suo codice per via della CFO.

Rimuovere uno strato di CFO è complesso. Nel tempo sono nati vari framework come Triton, Angr e Miasm per l'analisi concolica (crasi di Concreta e Simbolica) ma sono spesso poco o nulla documentati (con l'eccezione di Angr) e non sono comunque in grado di semplificare un blocco di istruzioni x86 o sbrigliare un CFG perchè, in generale, il problema è molto complesso.

Prima di affrontare il problema del districare il CFG del packer è quindi opportuno verificarne il rapporto fattibilità/sforzo.

Guardando meglio il grafico della funzione `WinMain`, si nota che è inizialmente composto da una catena di blocchi decisionali della forma mostrata sotto. Il ramo "Vero" della condizione porta ad un blocco di "codice di lavoro" che poi torna con un jump all'inizio della catena; il ramo "Falso" va invece ad un altro blocco.



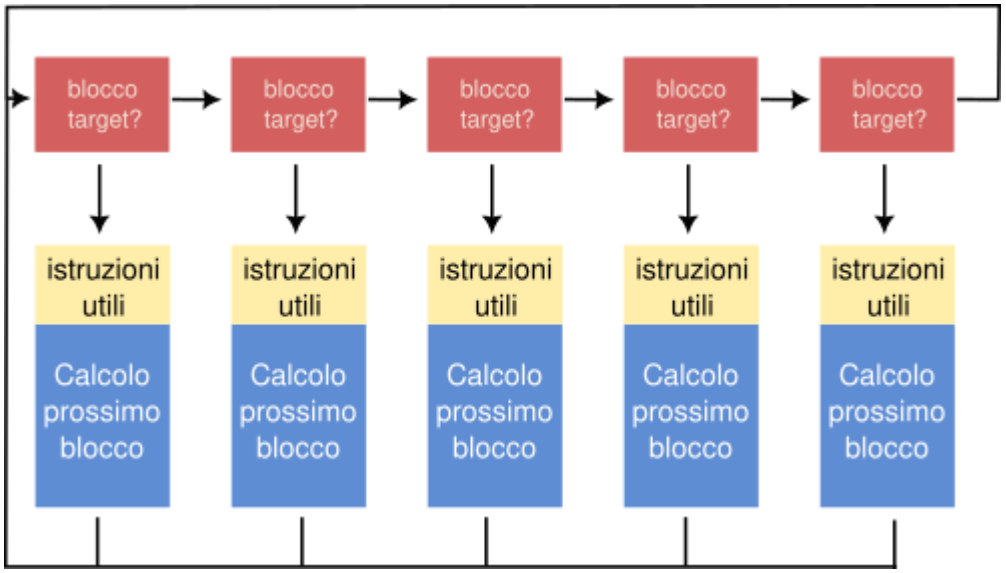
Un blocco decisionale del dispatcher

I blocchi di lavoro sono composti da istruzioni originali del packer e da una serie di istruzioni aritmetiche che calcolano un valore, detto ticket, per determinare il prossimo blocco di lavoro a cui saltare.

```
loc_403AEB:
xor     eax, eax
mov     ecx, cs:dword_431970
mov     edx, cs:dword_431974
mov     r8d, eax
sub     r8d, 1
add     r8d, 0
mov     r9d, eax
sub     r9d, ecx
mov     r10d, eax
sub     r10d, r8d
add     r9d, r10d
sub     eax, r9d
imul    ecx, eax
and     ecx, 1
cmp     ecx, 0
setz    r11b
cmp     edx, 0Ah
setl    bl
mov     sil, r11b
xor     sil, 0FFh
mov     dil, bl
xor     dil, 0FFh
mov     r14b, 1
xor     r14b, 1
or      sil, dil
or      r14b, 1
xor     sil, 0FFh
and     sil, r14b
mov     dil, r11b
xor     dil, 0FFh
mov     r14b, bl
and     r14b, dil
xor     bl, 0FFh
and     r11b, bl
or      r14b, r11b
mov     r11b, sil
and     r11b, r14b
xor     sil, r14b
or      r11b, sil
test    r11b, 1
mov     eax, 8858DF56h
mov     ecx, 2FF783BCh
cmovnz  ecx, eax
mov     [rbp+350h+pilot1], ecx
jmp     loc_4100C3
```

Le istruzioni per il calcolo del prossimo blocco.

Questa è una classica struttura a *dispatcher loop* che possiamo schematizzare come segue:



La struttura del dispatcher loop del packer

Per sbrogliare la matassa, che è al momento il CFG di `WinMain`, sono necessari i seguenti passi:

1. Ricreare il CFG di WinMain in Python per poterlo analizzare.
2. Determinare tutti i blocchi di dispatch (quelli rossi nell'immagine sopra) ed il valore del ticket a loro associato. Per fortuna hanno tutti un formato fisso con tre istruzioni (tranne il primo che è una leggera variante) e non sono presenti istruzioni superflue.
3. Analizzare il blocchi che calcolano il prossimo (o i prossimi) ticket e determinare questi valori.
4. Riscrivere il CFG collegando tra loro solo i blocchi di istruzioni utili.
5. Riassemblare le istruzioni in un PE.

Per gestire le istruzioni x86 abbiamo usato iced x86. Lo script finale è disponibile qui (lo script è pensato per un sample specifico). Descriviamo brevemente le parti più interessanti.

L'analisi e la trasformazione di codice di basso livello non è generalmente complicata ma è laboriosa: nel caso di codice x86 è proprio tediosa. Ogni singolo *mnemonic* ha svariate forme di codifica, senza contare che alcune istruzioni sono codificabili con *opcode* diversi (`add al, 1` potrebbe essere `add al, imm8` o `add rm8, imm8`). Inoltre la forma a due operandi delle istruzioni è più scomoda di quella a tre operandi per l'analisi (analogo discorso per il registro dei flag, che è un operando/output implicito). Infine il codice a 64 bit rende impossibile la trasformazione di operazioni aritmetiche lavorando direttamente sulle istruzioni perchè gli immediati sono rimasti a 32 bit e non è possibile unire due somme o due sottrazioni poichè il risultato potrebbe non essere rappresentabile come immediato.

I blocchi gialli delle istruzioni utili possono essere in realtà un grafo di blocchi ed è quindi necessario trovare i blocchi blu a partire a un blocco giallo. Per fortuna tutti i blocchi blu saltano ad un blocco unico che poi torna all'inizio del dispatcher.

Per la costruzione del CFG abbiamo usato la proprietà `flow_control` delle istruzioni iced x86: questa indica il tipo di control flow in seguito all'esecuzione dell'istruzione stessa e permette quindi di determinare facilmente la fine dei Basic Block (BB). L'unica operazione da tenere in considerazione è la possibilità per un BB di saltare nel mezzo di un BB precedente, che andrà quindi diviso. Ogni blocco ha un possibile blocco successivo (next) ed un

possibile blocco di salto (branch) a seconda del tipo di salto che lo termina (o nessuno dei due se, ad esempio termina con un `ret`). Quando si crea il CFG è utile ritornare il primo blocco ma anche una mappa che associ ogni VA in cui inizia un blocco con il blocco stesso.

La funzione che districa il flusso di esecuzione è mostrata qui sotto. Vengono subito identificati i nodi iniziali e finali del dispatcher loop (il primo è il secondo nodo della funzione, il secondo è il nodo che salta all'inizio). Il primo blocco contiene il valore iniziale del ticket, da cui è anche possibile capire l'offset rispetto ad `RBP` in cui è salvato.

```
def detangle(start: BasicBlock, context: typing.Dict[int, BasicBlock]): #Step 1: delimit the dispatcher loop dispatcher_start = start.next dispatcher_end = [bb for bb in context.values() if bb.next == dispatcher_start and bb != start][0] #Step 1b: find the initial ticket initial_ticket = None ticket_var_offset = None for i in start.instructions: if i.code == ix86.Code.MOV_RM32_IMM32 and i.memory_base == ix86.Register.RBP and i.memory_index == ix86.Register.NONE: initial_ticket = i.immediate(1) ticket_var_offset = i.memory_displacement break if initial_ticket is None: raise ValueError("Initial ticket not found") #Step 2: find all the dispatching branches #The form is: jmp -> mov, sub, jz (T)-> work code # (NT)-> next dispatching branch tickets = {None: None} blocks = [dispatcher_start] while blocks[-1].end == BasicBlockEnd.CONDITIONAL: cur_block = blocks[-1] for i in cur_block.instructions: if i.code in [ix86.Code.SUB_RM32_IMM32, ix86.Code.SUB_EAX_IMM32] and i.op0_register != ix86.Register.NONE: tickets[i.immediate(1)] = cur_block.branch if len(cur_block.next.instructions) == 1 and cur_block.next.end == BasicBlockEnd.UNCONDITIONAL: blocks.append(cur_block.next.next) #Step 3: Rewire start.set_next(tickets[initial_ticket]) todo = [start.next] done = set() while todo: block = todo.pop(0) if block is None or block in done: continue done.add(block) for b in dfs_reach_dispatcher_end(block, dispatcher_end.va): tn, tb = simplify(b, ticket_var_offset) b.set_next(tickets[tn]) b.set_branch(tickets[tb]) todo.append(b.next) todo.append(b.branch) def ilen(i, va): if i.len > 0: return i.len e = ix86.Encoder(64) return e.encode(i, va) #Step 4: layout the BBS va = start.va code = [] done = {} blocks = [(None, start)] while blocks: j, b = blocks.pop() if b in done: block_va = done[b] if j is not None: code[j] = ix86.Instruction.create_branch(code[j].code, block_va) else: code.append(ix86.Instruction.create_branch(ix86.Code.JMP_REL32_64, block_va)) va += ilen(code[-1], va) continue elif j is not None: code[j] = ix86.Instruction.create_branch(code[j].code, va) done[b] = va ni = b.new_instructions(ticket_var_offset) if len(ni) == 0: assert(b.branch is None) for i in ni: i.ip = va va += ilen(i, va) code.extend(ni) if b.branch is not None: blocks.append((len(code)-1, b.branch)) if b.next is not None: blocks.append((None, b.next)) encoder = ix86.BlockEncoder(64, False) encoder.add_many(code) encoded_bytes = encoder.encode(start.va) return encoded_bytes
```

I nodi rossi sono trovati partendo dal primo e seguendo i rami Falsi (ovvero i `next`, per la forma di salto) di ogni blocco, fino a che non si torna all'inizio. Mentre si attraversano i nodi rossi vengono collezionati anche i valori del ticket corrispondenti.

A questo punto abbiamo una mappa che per ogni ticket ci fornisce il blocco di istruzioni utili associato. Partendo dal ticket iniziale otteniamo il prossimo blocco di istruzioni utili. Tramite la funzione `dfs_reach_dispatcher_end` troviamo la lista di nodi blu raggiungibili. I nodi intermedi non sono analizzati in quanto non necessari. Ogni nodo blu raggiunto viene semplificato e dalle istruzioni risultati viene determinato il prossimo ticket. In alcuni casi il

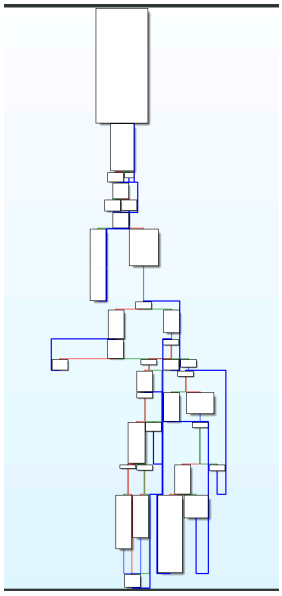
valore del prossimo ticket dipende da una condizione e per fortuna in questi casi il formato usato è sempre il solito e può essere gestito riconoscendolo esplicitamente. Ottenuti i possibili ticket del nodo blu corrente, modifichiamo i valori `next` e `branch` di questo e continuiamo l'analisi dai prossimi valori.

La semplificazione delle istruzioni consiste nelle seguenti operazioni ripetute in ciclo fino a che non vi sono ulteriori operazioni da svolgere:

- Riscrittura degli zero idiom (es: `xor eax, eax -> mov eax, 0`)
- Riscrittura delle load da indirizzi noti (es: `mov eax, dword_X -> mov eax, <dato ad X>`)
- Folding delle costanti da `mov reg1, imm1` ad istruzioni aritmetiche tipo `add reg1, imm2` o `add reg2, reg1` o a store su variabili locali.
- Rimozione di dead stores.
- Riscrittura di `imul reg1, reg2` se reg1 e reg2 hanno valore costante.
- Merge di istruzioni `add` e `sub` sul solito registro.
- Riscrittura di `cmp/setcc` e `test/cmovcc` .

L'implementazione di queste ottimizzazioni richiede un po' di lavoro ma permette di trasformare il blocco blu mostrato ad esempio sopra (che contiene 45 istruzioni) nella singola istruzione `mov dword ptr [rbp-38h],8858DF56h` !

Una volta riscritto il PE (abbiamo ignorato questioni legate allo spazio e rilocazioni, trattandosi di un PE a 64 bit) il CFG ottenuto è decisamente più semplice:



Il flusso di esecuzione del packer una volta deoffuscato

Sebbene le istruzioni che effettuano operazioni a 64 bit non siano state semplificate è comunque possibile intravedere il comportamento del packer. E' innanzi tutto interessante notare come venga usato yandex.com per determinare se la vittima ha accesso ad internet:

Controllo di connettività tramite yandex.com

Analizzando il codice, si vedono le istruzioni con le quali il packer recupera i dati dalla sezione `.data` e queste confermano quanto scoperto tramite la crittoanalisi elementare:

Codice che conferma che la chiave ha lunghezza 20 byte e si trova dopo la dimensione del payload.

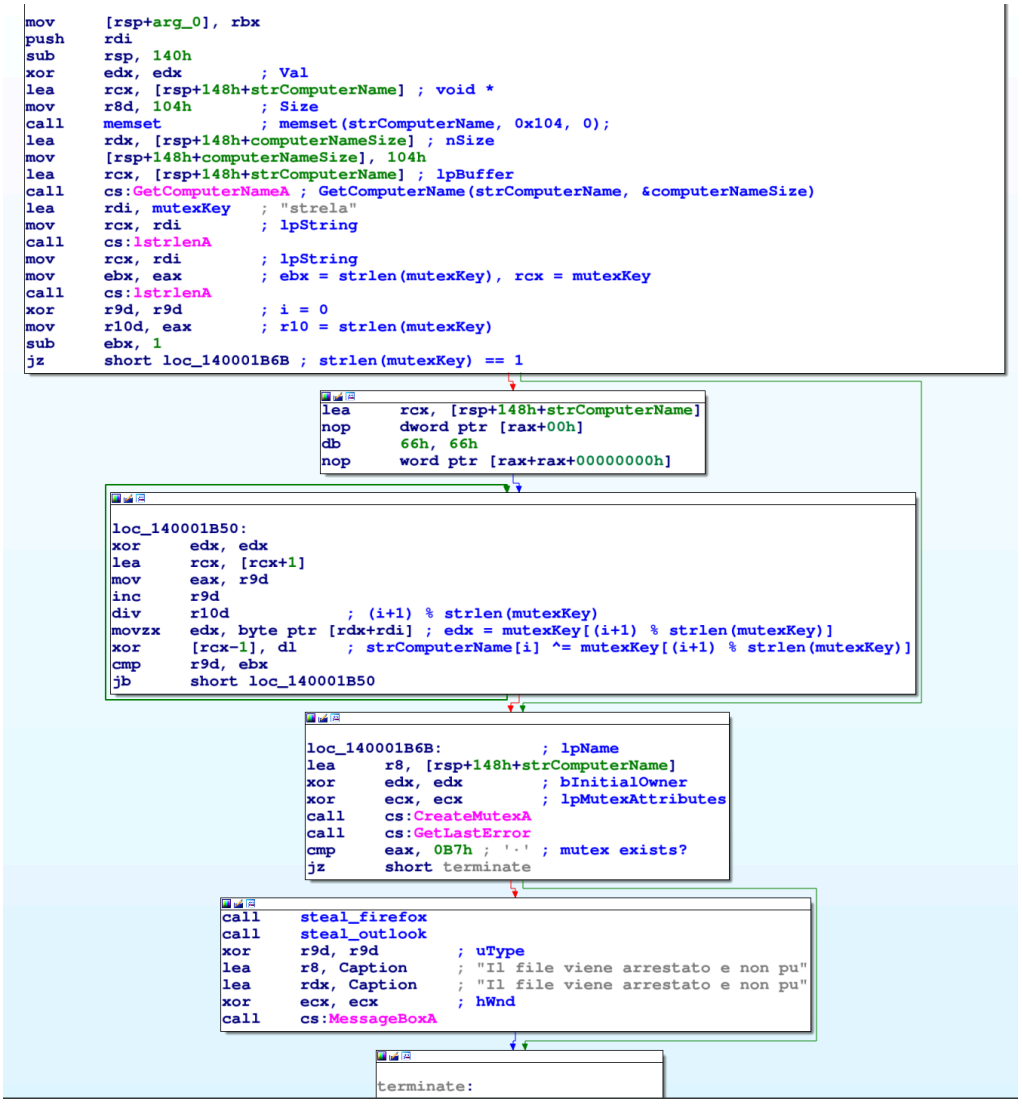
Questo tipo di analisi statica è piuttosto laboriosa ed è generalmente fattibile solo in una seconda fase. Ha l'innegabile vantaggio di mostrare il vero codice del malware in oggetto, spesso necessario per determinare le TTP usate e non cadere vittima di comportamenti decoy.

Il malware Strela

A discapito della complessità del packer, il malware Strela è estremamente semplice.

Strela è un malware minimale che ha come uniche funzionalità quelle di rubare gli account di posta da Thunderbird ed Outlook.

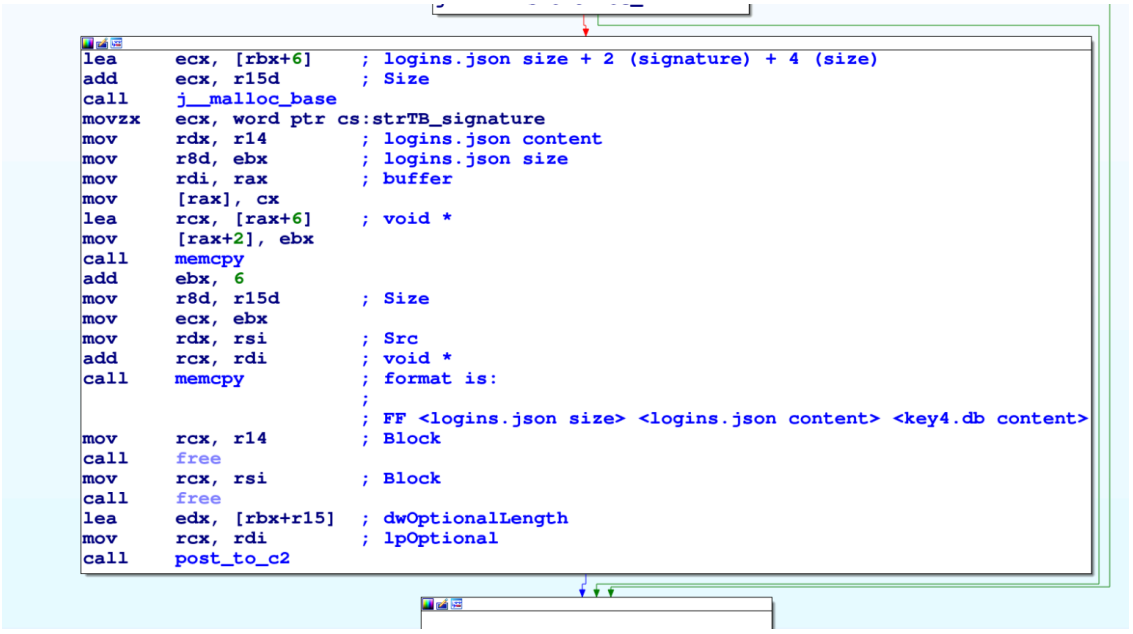
Non presenta alcun tipo di offuscazione, il C2 è in chiaro nel malware, così come tutte le altre stringhe. Il codice della funzione principale del malware sta tutto nell'immagine sotto:



Il codice di Strela

Il malware si assicura che una sola istanza sia in esecuzione tramite un mutex (il cui nome è dato dal nome del computer della vittima XORato con la stringa *strela*, da cui il nome del malware). Dopodichè ruba gli account di Thunderbird ed Outlook, mostra un messaggio di errore all'utente (in un italiano non proprio idiomatico) e termina.

Il codice della funzione `steal_firefox` mostra come Strela cerchi il primo profilo Thunderbird tramite `FindFirstFileA` e da questo legga i file `logins.json` e `key4.db`. Il seguente blocco di codice crea il buffer da inviare al C2.



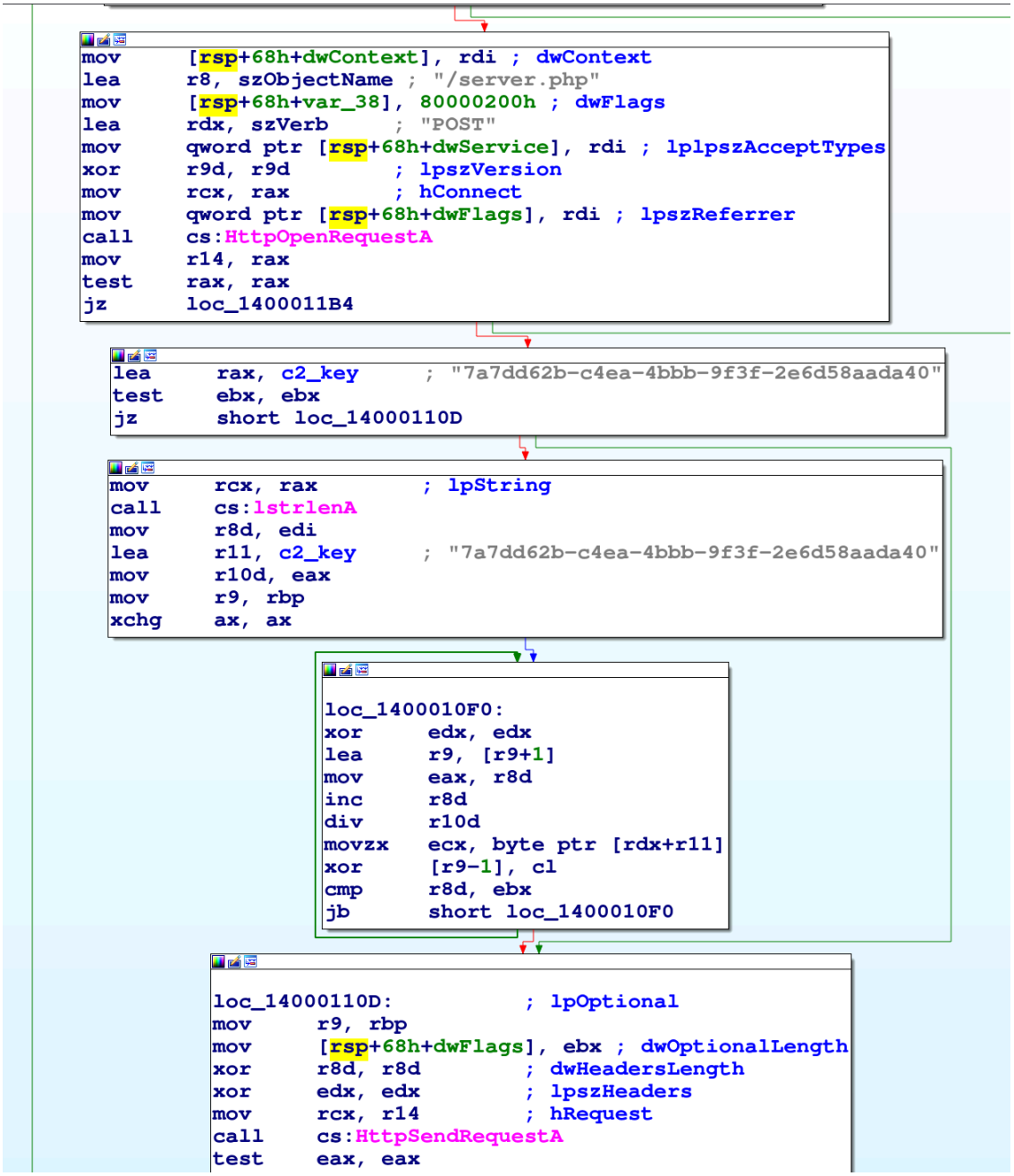
Il formato del buffer con i dati di Thunderbird.

Il formato usato è `<firma><dimensione logins.json><dati logins.json><dati key4.db>`, dove firma è a stringa *FF* e la dimensione è una DWORD (in little endian).

Per i dati di Outlook il formato è ancor a più semplice: dopo la firma *OL* è presente una serie di righe `<server>, <username>, <password>`.

Formato dei dati inviati al C2

Una volta ottenuto un buffer da inviare al C2, Strela usa la chiave `7a7dd62b-c4ea-4bbb-9f3f-2e6d58aada40` per cifrarlo tramite rolling XOR. Il risultato è inviato tramite POST all'URL *http ://91 . 215 . 85. 209 / server .php*.



Se il formato è corretto, il C2 risponde con una sequenza casuale di byte terminata dalla stringa *KH*.

Il C2

Il C2 è localizzato in Russia, apparentemente di proprietà di una società di nome Prospero registrata nel novembre del 2022 e di cui non si trovano servizi online (forse è un servizio di hosting). Da una rapida indagine OSINT si trovano evidenze che sullo stesso ASN sono stati ospitati, dai primi del 2023, C2 di altre campagne malware.

Inviando al C2 un pacchetto corretto, questo risponde con una sequenza casuale di byte terminati dalla stringa *KH*.

```
bytearray(b"\xd2m\x94\xfbO\x0e\xd3\x90\xf7\x01\xc6\x17l\xa5%f\xce\xadS\x16\x8eg\xe8\xcc7\xb4\xfe\xe3K\xc2\x8d\x19<\x
c8\xde\xa1-\xa1p\xc8\xa4\xd60Zt\xac\x1c+\xaeI\xf2H\xb4\x16\xd0\x11\x10o:\xa8GN\xcc\x12\xd4U\xf6o\x8a\'\xdd\xcb\x15\x
91k\xc4\xc0\xbe\xeb\x90\x0b\x8f<6%\x9ct\xcfU\xc8\xe9\xe4d\x1b`\xde\x03o\x8c\x8f\x17i\xd4\xbb\n\x0biu6\xf4_c\xf1\x87O
\xaa\x08g\x05\xca\xe4\xeb\x7f\xe1\xa0HKH")
```

La risposta del C2 ad un pacchetto valido.

Se invece inviamo un pacchetto non valido, la risposta è vuota.

Ma cosa constituisce un pacchetto valido? Consideriamo il pacchetto per l'invio dei dati di Thunderbird, poichè è più complesso. Ovviamente devono essere presenti la firma e la dimensione del primo file (logins.json) ma… sono eseguiti ulteriori controlli?

Da una rapida serie di tentativi si evincono le seguenti evidenze:

1. Il C2 ritorna una risposta vuota se il contenuto di logins.json non è un JSON valido.
2. Il C2 ritorna una risposta vuota se il contenuto di key4.db non è un DB SQLite valido.
3. Il C2 ritorna una risposta vuota se il database key4.db non contiene dati.

Il terzo punto ed il primo punto sono interessanti perchè indicano che il C2 processa i dati online. Era auspicabile che i file venissero semplicemente salvati, invece sono processati al momento della loro ricezione e l'esito viene ritornato al client sotto forma di risposta vuota (in caso di errore) o meno.

Questo permette di avere un *oracolo* e, associato al fatto che vengono effettuate delle query su un DB SQLite, permetterebbe di recuperare alcune informazioni.

Idealmente, si potrebbe seguire una linea simile a quella delineata dall'ottimo articolo sul QOP di Checkpoint, ma le recenti versioni di SQLite non hanno vulnerabilità usabili come oracolo e inoltre la versione SQLite3 usata è la 3.34.1 per la quale non valgono le CVE identificate nell'articolo di Checkpoint. Rimane il fatto che l'azione del C2 di fare query su un DB controllato dal client possa comunque essere un punto di partenza per ulteriori analisi in seguito.
