Tecniche per semplificare l’analisi del malware GuLoader

cert-agid.gov.it/news/malware/tecniche-per-semplificare-lanalisi-del-malware-guloader/

21/07/2022

guloader

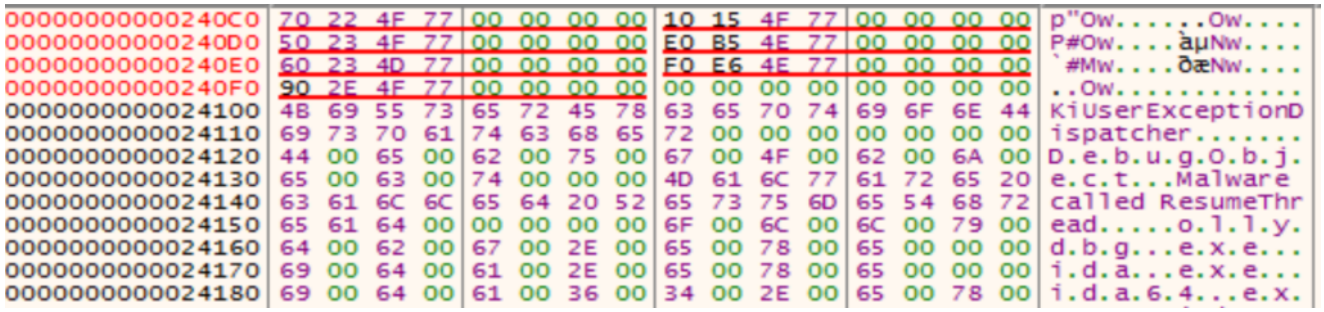
Gli analisti di CERT-AgID hanno osservato GuLoader in Italia per la prima volta verso la fine mese di marzo 2021. Nell’arco dello scorso anno sono state registrate solo 6 campagne che utilizzavano GuLoader sfruttando il tema “*Pagamenti*“, “*Preventivo*” e “*Ordine*” con lo scopo di veicolare il malware **AgentTesla** ed in un solo caso si è avuta evidenza del rilascio di **Remcos**.

Le campagne GuLoader in Italia sono terminate a fine settembre 2021 per poi ripresentarsi nel 2022, mantenendo gli stessi temi, con 4 nuove campagne: una ad aprile, un’altra a metà giugno e le ultime due – ad un mese esatto di distanza – a metà luglio.

GuLoader è un dropper che si caratterizza per l’efficacia delle sue misure anti-debug e anti-vm. Il CERT-AgID aveva già discusso la natura di tali misure, anche se al tempo il packer non era stato identificato come GuLoader. Ad oggi, tali tecniche sono state affinate e ve ne sono state aggiunte di nuove, al punto che analizzare GuLoader è diventato un compito abbastanza complesso.

Due vecchie tecniche migliorano

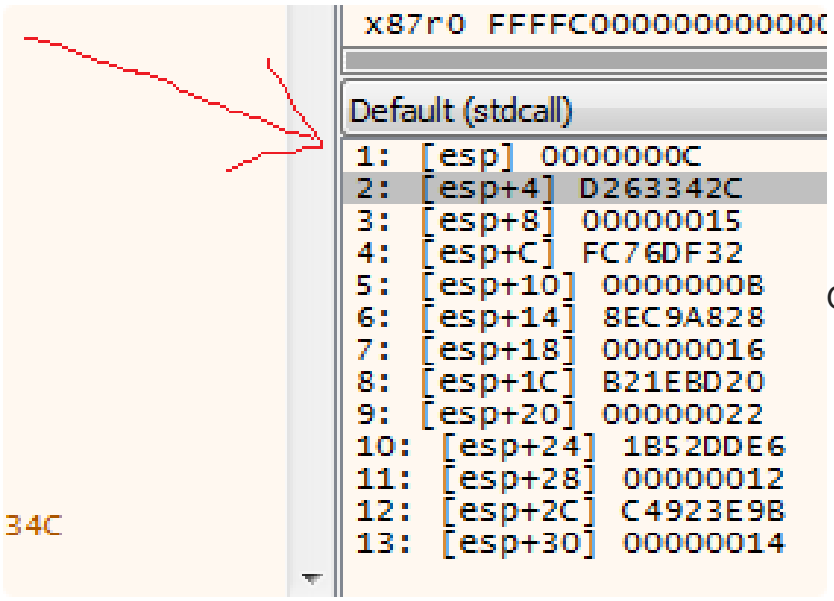
GuLoader disponeva di un controllo anti-debug che, anzichè cercare un comportamento anomalo delle API di Windows, **verificava la presenza in memoria di artefatti usati dai debugger** (o loro plugin) per nascondersi dai malware.



Esempio di un’area di memoria contenente codice e dati di Scylla, un plugin per nascondere il debugger, in un processo sotto debug.
GuLoader non controllava direttamente la presenza di valori specifici ma utilizza l’hashing DBJ2 su gli indirizzi di porzioni di memoria determinate empiricamente.

Questa tecnica era totalmente efficace nel rilevare i debugger più usati: la chiamata alla funzione che effettuava questo controllo era **facilmente identificabile** per via del fatto che prendeva un gran numero di argomenti (gli hash degli artefatti) terminati dal valore `0xffffffff`. In questo caso era sufficiente rimpiazzare la chiamata con dei `NOP` per superare l'ostacolo.

Questa tecnica, oggi, esiste ancora ma qualcosa è stato cambiato. Gli argomenti passati alla funzione non si limitano agli hash ma contengono anche dei numeri.



Gli argomenti della funzione che controlla la presenza di debugger. Sono coppie composte da un numero ed un hash, terminate da uno zero.
Probabilmente la tecnica di rilevamento è stata aggiornata per essere più resiliente ed adeguarsi alla continua evoluzione dei debugger.

Ancora oggi il rilevamento del debugger è efficace e l'unica opzione per eludere questo controllo è quello di individuare la chiamata e di rimpiazzarla, oppure disinstallare i plugin nella speranza che il debugger non abbia artefatti propri rilevati da GuLoader. Disabilitare i plugin però comporterà la facile individuazione del debugger tramite le usuali tecniche anti-debug (es: tramite *NtQueryInformationProcess*) che GuLoader non disdegna. Per debuggare GuLoader è quindi necessario procedere passo passo fino all'individuazione di questa chiamata. Tuttavia, gli autori del dropper hanno individuato un metodo per rendere l'analisi passo passo molto tediosa.

Tecniche di Anti-VM

Sfortunatamente per noi, GuLoader ha un ottimo controllo anti-vm che continua ad ingannare anche le sandbox online. Quindi, eseguirlo in una VM insieme ad uno strumento in grado di monitorare il traffico di rete non è sufficiente per ottenere il drop URL ed il payload.

La tecnica che veniva usata nel campione analizzato nel bollettino allegato era stata battezzata *RDSTC trick* e si basava su un assunto molto semplice: l'istruzione `cpuid` causa un VM-exit non condizionale ed il suo risultato deve essere alterato dall'hypervisor (poichè descrive le caratteristiche e le estensioni della CPU): questo comporta che **in un ambiente virtualizzato la sua esecuzione sia più lenta che in uno fisico**. Per effettuare questo genere di misurazioni è necessario un timer molto preciso ed a bassa latenza di accesso, il timestamp counter (detto anche TSC e letto tramite `rdtsc`) presente nelle CPU Intel e compatibili è l'ideale. **La parte complessa è tarare bene le soglie di rilevamento**.

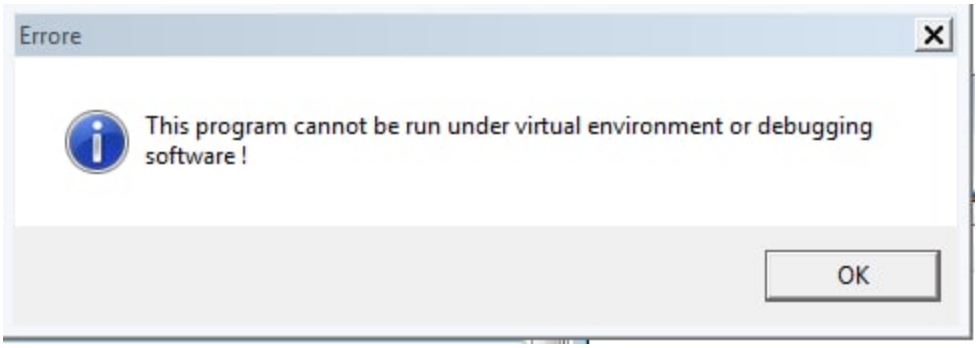
> **Nota tecnica**
>
> Nelle CPU moderne il TSC è un contatore slegato dalla frequenza e dallo stato energetico della CPU. Tuttavia, lo stato energetico (si vedano le tecnologie di gestione termica di Intel, da SpeedStep a HWP passando per Turbo Boost) della CPU influenza pesantemente il tempo cronometrato di esecuzione delle istruzioni, per cui misurare la durata delle istruzioni con il TSC non è molto affidabile ma probabilmente sufficiente agli scopi.

Nel campione attuale questo controllo è stato stravolto.

A sinistra: il controllo sulla durata di cpuid e rdtsc tramite il timer di Windows. A destra: il controllo che rdtsc non ritorni valori fasulli.
Un workaround per l'*RDTSC trick* era quello di emulare un TSC lento.

La nuova strategia di temporizzazione utilizza il timer di Windows, accede direttamente ad user space tramite KUSER_SHARED_DATA e misura l'esecuzione di `cpuid` e `rdtsc` ripetutamente. Qualora il valore accumulato superi una certa soglia, GuLoader assume di trovarsi in presenza di una VM. Viene aggiunto anche un controllo esplicito che verifica se rdtsc ritorna valori falsi, ad esempio che siano troppo "lenti".

Questi controlli sono efficaci e portano GuLoader a mostrare una finestra di avviso e terminare o entrare volutamente in un ciclo infinito, prevenendo l'analisi automatica.



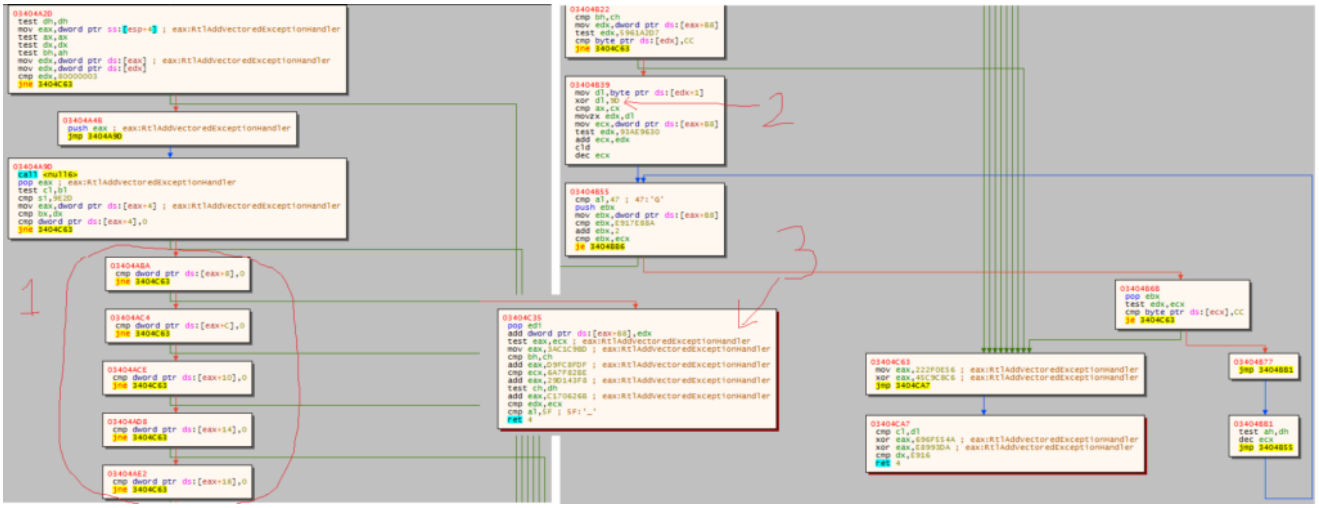
In aggiunta a questi controlli di temporizzazione è sempre presente la verifica del bit 31 di `CPUID.1.ecx`, che **indica la presenza di un hypervisor con supporto di paravirtualizzazione**. Dato che le VM non tendono a nascondersi, questo controllo risulta efficace. Disabilitare la paravirtualizzazione ha i suoi costi rendendo l'esecuzione della VM più lenta ed onerosa.

GuLoader cerca inoltre di determinare se è eseguito dentro una VM anche tramite metodi più convenzionali. In particolare utilizza `EnumDeviceDrivers` e `EnumServicesStatusA` per enumerare i driver ed i servizi tipicamente installati nelle VM paravirtualizzate (es: *vmmouse.sys*).

Anche queste misure sono piuttosto efficaci nel rilevare le VM. Maggiori dettagli sono riportati in questa analisi di SonicWall.

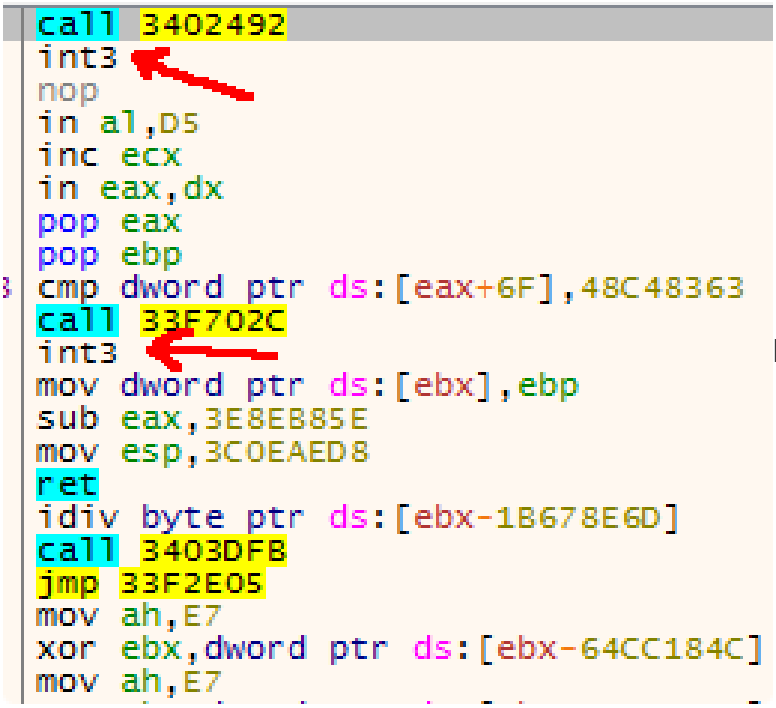
Tecnica Anti-Analisi

Nonostante la presenza di questi controlli, inizialmente era più semplice riconoscerli e saltarli. Oggi questo è diventato più complesso per via di una tecnica anti-analisi introdotta da qualche mese e piuttosto fastidiosa.



L'exception handler che sposta l'instruction pointer dopo ogni istruzione int3. Il numero di byte di cui spostare eip in avanti è ottenuto come xor tra 0x9d ed il byte successivo ad int3.

Subito dopo aver decifrato il suo secondo stadio, lo shellcode di GuLoader installa un exception handler tramite `RtlAddVectoredExceptionHandler`. Questo handler è invocato tramite delle istruzioni *int3* sparse in tutto il codice.



Istruzioni int3 sparse per il codide del dropper. Come si intuisce dalla presenza di istruzioni privilegiate, l'esecuzione non è lineare. Come mostra il codice qui sopra, questo handler ha due funzioni:

1. Verifica che non siano presenti **breakpoint** software (dopo l'istruzione `int3`) o hardware.
2. Legge il valore del **byte successivo** all'istruzione `int3`, effettua uno *xor* con `0x9D` e aggiunge questo valore all'instruction pointer, di fatto spostando l'esecuzione in avanti.

I controlli anti-debug di cui il punto 1) possono essere disabilitati rimpiazzando i salti condizionali con dei *nop*. Ma il secondo punto rimane problematico: il debugger decodificando le istruzioni sequenzialmente si confonde e diventa impossibile avere una visione d'insieme del codice, rendendo complesso il riconoscimento delle funzioni. Infine, quando l'handler ritorna con il valore `EXCEPTION_CONTINUE_EXECUTION` l'esecuzione torna al codice interrotto tramite `NtContinue`, la quale non da modo al debugger di interrompere immediatamente il processo, di fatto facendo saltare l'analisi "da *int3* in *int3*".

Per aggirare il problema di non controllo sull'esecuzione è necessario ricorrere a degli script per il proprio debugger. Ad esempio, per *x64dbg* è possibile usare le seguenti istruzioni (quando `eip` è su `int3`):

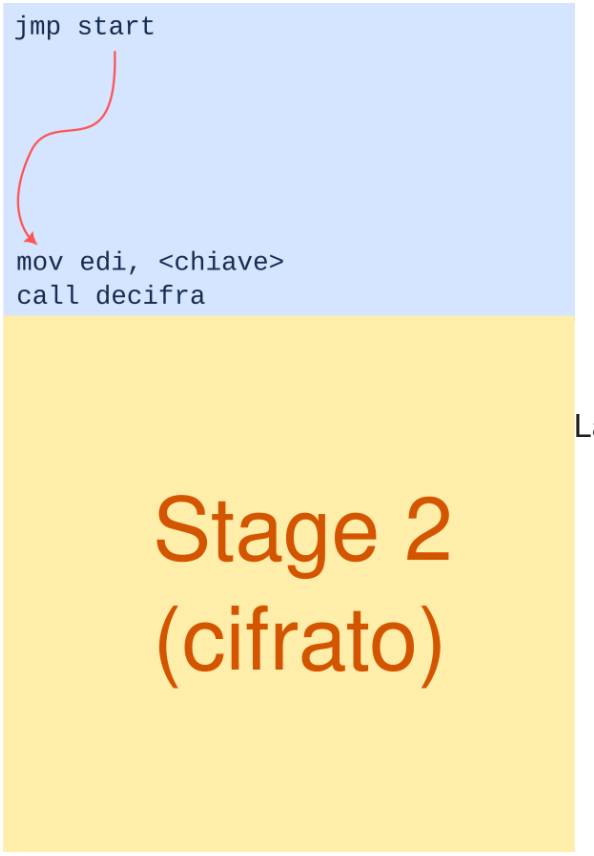
```
$ec = byte(eip + 1); xor $ec, 0x9d; eip = eip + $ec;
```

Estrarre il drop url automaticamente

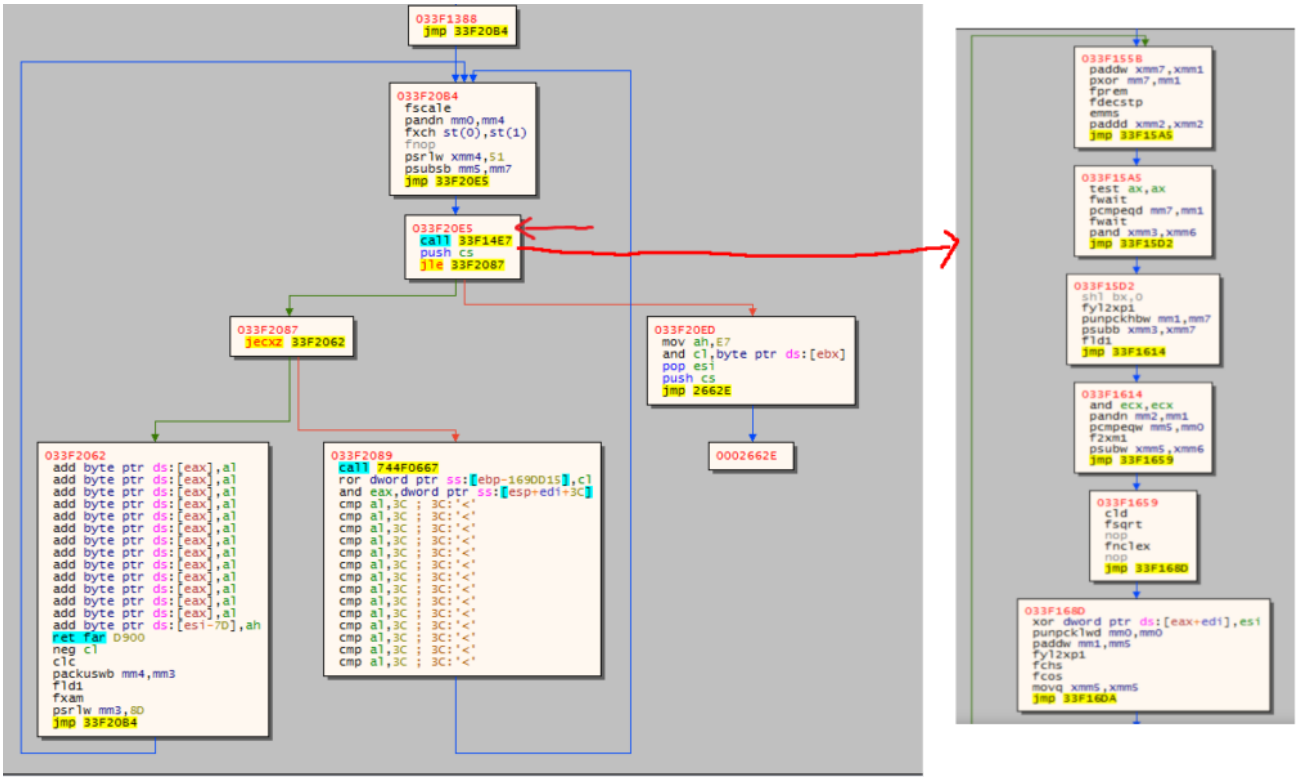
La nuova tecnica anti-analisi di GuLoader rende il debug molto tedioso e la presenza di numerosi controlli anti-vm ed anti-debug non permettono l'esecuzione non controllata del dropper.

È possibile velocizzare l'analisi?

Lo shellcode di GuLoader appena avviato salta ad una procedura che decodifica il secondo stadio. La struttura dello shellcode è la seguente:

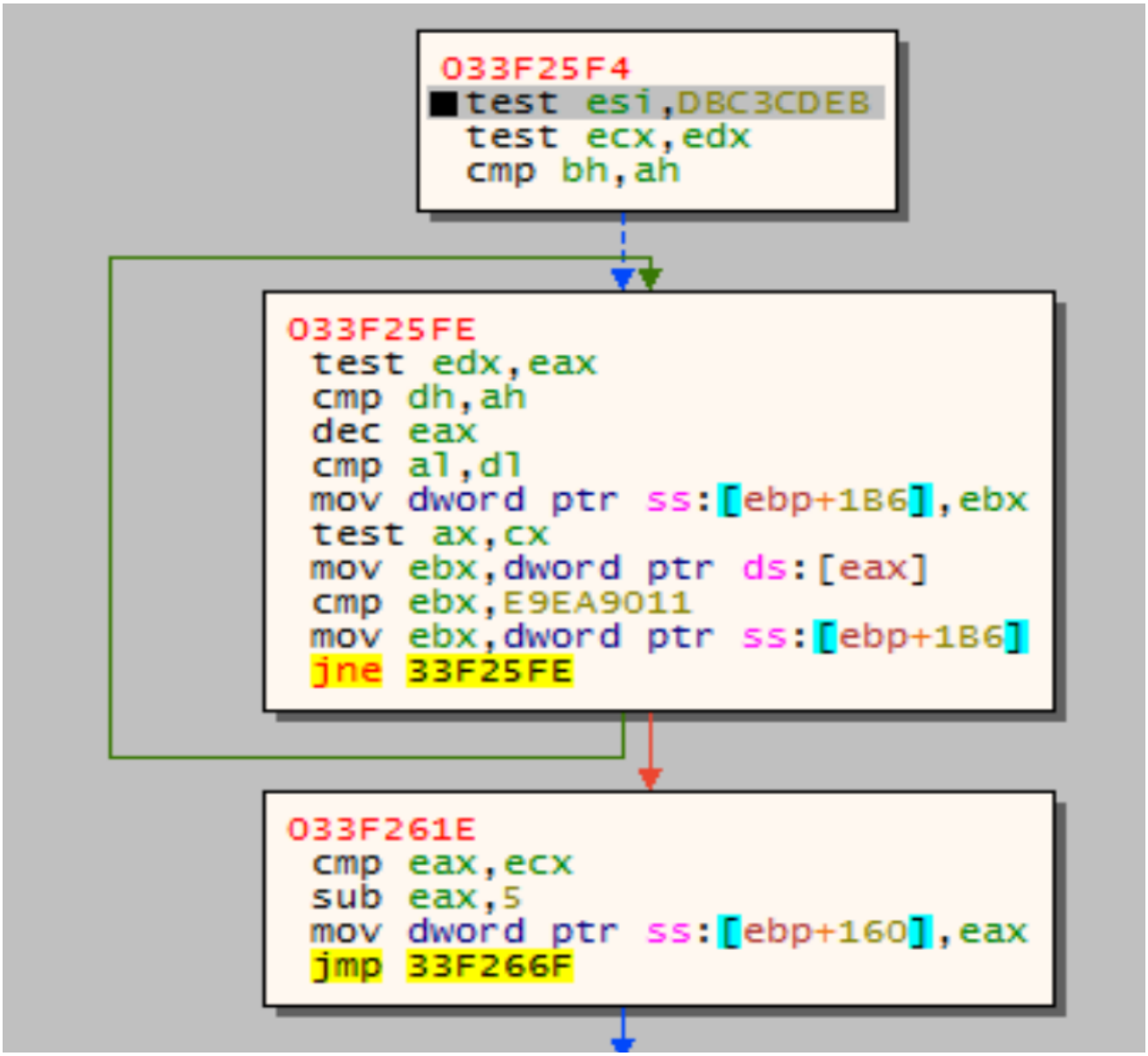


La funzione che decifra il secondo stadio si trova subito prima di esso.
C'è una prima parte, in blu, che non viene decodificata: essa contiene il codice di decodifica stesso. Tale codice è chiamato tramite un'istruzione `call` situata subito prima dell'inizio del secondo stadio. Questo fa sì che dentro tale chiamata l'**indirizzo di ritorno punti proprio al secondo stadio**.



Decodifica del secondo stadio tramite xor. La freccia verso sinistra indica l'istruzione *call* subito prima del secondo stadio. A destra il codice di decodifica.

Nel secondo stadio, dopo una piccola pausa implementata con un ciclo che esegue `rdtsc`, GuLoader determina l'inizio del secondo stadio cercando la DWORD *0xE9Ea9011*.



Guloader determina l'inizio del secondo stadio cercando la DWORD 0xE9Ea9011 e sottraendovi 5 (la lunghezza di un *jmp* lungo).
Possiamo ipotizzare che la chiave di decodifica vari da campione a campione: uno strumento automatico dovrebbe essere in grado di estrarla o calcolarla. Estrarla è complesso perchè è generata tramite istruzioni aritmetiche e richiederebbe l'esecuzione concolica (simbolica + concreta) dello shellcode. Analogo discorso per l'inizio del secondo stadio.

Un'alternativa è quella di sfruttare le debolezze della cifratura con *xor* e chiave piccola.

Il secondo stadio probabilmente conterrà delle sequenze di byte nulli: queste sequenze rilevano la chiave ma il tutto sta nel capire dove sono. Piuttosto che utilizzare offset fissi, un approccio ragionevole è quello di considerare lo shellcode come una sequenza di DWORD (interi senza segno a 32 bit) ed ordinarli dal più frequente a quello meno frequente.

Ipotizziamo che tra i primi valori sia presente anche la rotazione della chiave. Parliamo di rotazione della chiave perchè lo *xor* può non iniziare ad indirizzi multipli di 4 bytes, ovvero: non è allineato a DWORD e in questa campione non lo fa.

Possiamo verificare velocemente questa ipotesi con un po' di codice Python. La chiave usata nel sample in analisi è *0xb49be733*.

```
def count_dwords(data, skew=0): hist = {} for i in range(0 + skew, (len(data)-skew)//4 * 4, 4): dw = struct.unpack("<I", data[i:i+4])[0] if dw <= 0xffffff: continue if dw not in hist: hist[dw] = 1 else: hist[dw] += 1 return {x[0]:x[1] for i, x in enumerate(sorted(hist.items(), key=lambda x: -x[1])) if i < 10}
```

Il risultato di `count_dwords` mostra che la (rotazione) della chiave è il primo risultato:

```
0x33e7b49b 53
0xff000000 39
0xbae7b49a 37
0xffe7b49a 35
0xfbfbfbfb 26
0x78787878 24
0xbae7b499 24
0x49494949 23
0xe5e5e5e5 23
0x74747474 23
```

Per ogni possibile chiave, possiamo fare lo *xor* con lo shellcode, includendo la parte blu, visto che non sappiamo dove finisce, e verificare la presenza del valore *0xE9EA9011*, esattamente come fa GuLoader. Questo ci permette **non solo di confermare che la chiave è giusta ma anche di determinare dove inizia il secondo stadio** in modo da decifrare soltanto quello.

```
def count_dwords(data, skew=0): hist = {} for i in range(0 + skew, (len(data)-skew)//4 * 4, 4): dw = struct.unpack("<I", data[i:i+4])[0] if dw <= 0xffffff: continue if dw not in hist: hist[dw] = 1 else: hist[dw] += 1 return {x[0]:x[1] for i, x in enumerate(sorted(hist.items(), key=lambda x: -x[1])) if i < 10} def xor(b1, i1, b2, i2, l, dbg=False): res = [0] * l for i in range(i1, i1+l): if dbg and i > len(b1): print("Wrap around1") if dbg and i-i1 < 16: print(hex(i % len(b1)), hex(b1[i % len(b1)]), hex((i2 + i-i1) % len(b2)), hex(b2[(i2 + i-i1) % len(b2)]), hex(b1[i % len(b1)] ^ b2[(i2 + i-i1) % len(b2)])) res[i-i1] = b1[i % len(b1)] ^ b2[(i2 + i-i1) % len(b2)] return bytes(res) def find_stage2(data, key): sign = b"\x11\x90\xea\xe9" #Alternative signature: b"\xe9\x4d\x01\x00" sign_off = 5 #Alternative offset: 0 dec = b"" for i in range(0, len(data), 4): dec = (dec + xor(data, i, key, 0, 4))[-8:] if sign in dec: j = dec.index(sign) return i+j-4-sign_off def shift(data, val): return data[val:] + data[:val] def decrypt_stage2(data): for k, v in count_dwords(data).items(): print(f"🤞
```

```
Possible (rotated) decrypt key: {hex(k)}") key = struct.pack("<I", k) offset = find_stage2(data, key) if offset is None: print(f"😐 No stage found for this key, trying next one.") continue else: print(f"Stage 2 found at offset {hex(offset)}") key = shift(key, offset & 0x3) print(f"Adjusted key to: {hex(struct.unpack('<I', key)[0])}") dec_data = data[:offset] + xor(data, offset, key, 0, len(data)-offset) print(f"  Stage 2 decrypted.") return dec_data
```

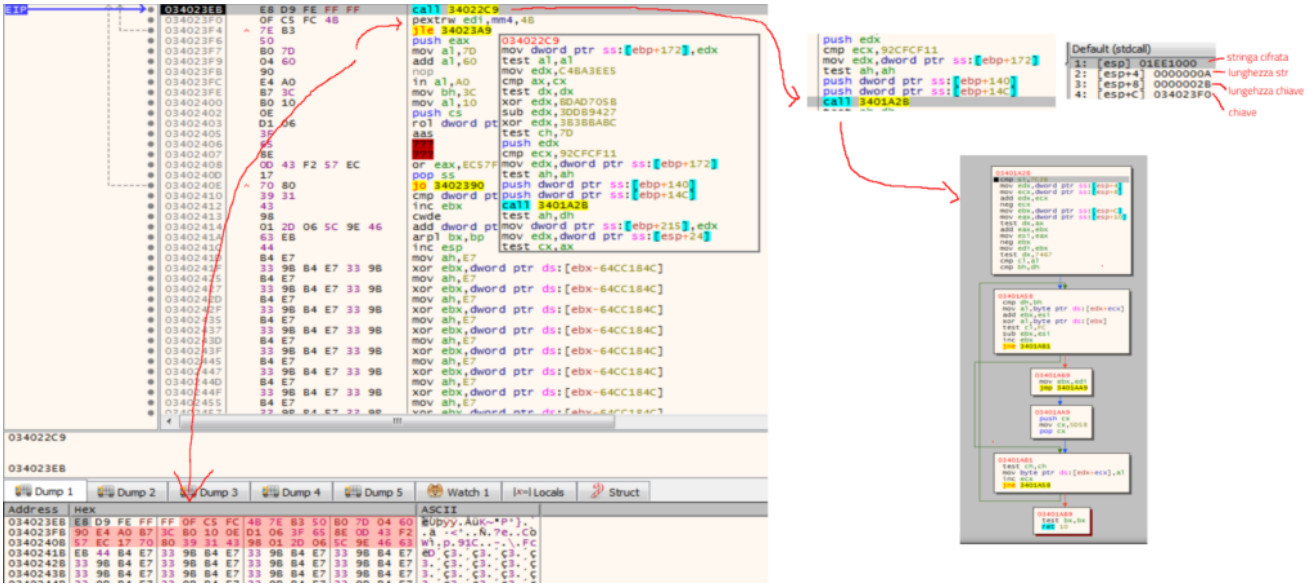
Nel campione analizzato il secondo stadio inizia a *0x20ea*. Il risultato dello script Python conferma che la decifratura è corretta:

```
✌️ Possible (rotated) decrypt key: 0x33e7b49b
Stage 2 found at offset 0x20ea
Adjusted key to: 0xb49b33e7
  Stage 2 decrypted.
```

Come possiamo usare il codice del secondo stadio per velocizzare l'analisi?

Il drop url è contenuto in una stringa codificata. Fortunatamente la prima azione di GuLoader, dopo aver determinato l'inizio del secondo stadio, è decodifcare la stringa *L"ntdll"* per cui possiamo subito analizzare come avviene questo processo.

Tenere traccia degli indirizzi è tedioso per via del codice superfluo: le stringhe sono salvate *xorate* con una chiave di *0x2b* byte e precedute da una DWORD che indica la lunghezza, anch'essa è *xorata* con una costante. Gli offset dove trovare queste stringhe codificate sono probabilmente fissi e generati tramite istruzioni aritmetiche per cui ottenerli è complicato.



Da sinistra a destra: La funzione che ottiene l'indirizzo della chiave tramite il proprio indirizzo di ritorno, la funzione intermedia che passa i parametri alla vera procedura di decifratura, il codice di decifratura.

Tuttavia, **se avessimo la chiave, potremmo provare un bruteforce alla ricerca di stringhe stampabili e, tra queste, quelle che iniziano per *http* o contengono *://***.

Con un po' di pazienza si trova facilmente che GuLoader ottiene l'indirizzo della chiave per decifrare le stringhe in modo analogo a come ottiene l'indirizzo del secondo stadio: ovvero **tramite una chiamata posizionata subito prima della chiave**. Con un po' di debug si scopre che la lunghezza di questa chiave è *0x2b* byte.

Come trovare la chiave nel secondo stadio decifrato?

L'idea è di cercare tutte le chiamate con opcode *0xe8* e offset negativo (salto all'indietro) e considerare i byte successivi come la chiave. La speranza è che non ve ne siano molte. In realtà possiamo provare a cercare esattamente i byte *0xe8, 0xd9, 0xfe, 0xff* se ipotizziamo che la distanza tra la chiave e la funzione di decifratura non cambi ed eventualmente tornare ad un metodo bruteforce nel caso questo fallisca.

Ottenuta la chiave è possibile fare un bruteforce su ogni offset e prendere le stringhe stampabili di almeno *n* caratteri. Si deve porre attenzione al fatto che le stringhe sono, o potrebbero essere, in UTF-16.

Nello script di seguito riportato vengono ricercate tutte le stringhe ma mostrate solo quelle con *http* o *://* ed è possibile ottimizzarlo per cercare solo quelle di interesse:

```
def get_string_key(data): call_strdec = b"\xe8\xd9\xfe\xff\xff" #The string key are the 0x2b bytes after this call. #If this fails, we can try looking for all E8 (relative) calls if call_strdec in data: i = data.index(call_strdec)+5 return data[i:i+0x2b] def find_strs(data, skey, mlen=100): strs = [] i = 0 while i < len(data): possible = xor(data, i, skey, 0, mlen) s = b""; for j in range(len(possible)): if (possible[j] >= 0x20 and possible[j] <= 0x7f) or possible[j] in [0xa, 0xd, 0x00, 0x07]: s += bytes([possible[j]]) else: break s2 = s.replace(b"\x00", b"") if len(s2) >= 5: strs.append(s2) i += len(s) i += 1 return strs def interesting_str(strs): res = False for s in strs: if b"http" in s or b"://" in s: print(s) res = True return res
```

Lo script completo

Lo script seguente è un PoC su come estrarre il drop url da un campione GuLoader. Potrebbe essere necessario sistemare `get_string_key` con una nuova firma o un'euristica. La nuova firma è ottenibile con una breve analisi: è possibile anche posizionare un breakpoint in `ZwAllocateVirtualMemory` e poi seguire le chiamate per arrivare direttamente alla funzione che decifra le stringhe (come mostrata nelle figure precedenti).

Lo script si esegue passandogli lo shellcode di GuLoader: questo va estratto manualmente dal vettore di infezione. Il campione in analisi utilizzava uno script NSIS per questo:

```
import struct from binascii import hexlify import sys def
read_shellcode(filename): with open(filename, "rb") as f: data = f.read()
return data def count_dwords(data, skew=0): hist = {} for i in range(0 +
skew, (len(data)-skew)//4 * 4, 4): dw = struct.unpack("<I", data[i:i+4])[0]
if dw <= 0xffffff: continue if dw not in hist: hist[dw] = 1 else: hist[dw]
+= 1 return {x[0]:x[1] for i, x in enumerate(sorted(hist.items(), key=lambda
x: -x[1])) if i < 10} def xor(b1, i1, b2, i2, l, dbg=False): res = [0] * l
for i in range(i1, i1+l): if dbg and i > len(b1): print("Wrap around1") if
dbg and i-i1 < 16: print(hex(i % len(b1)), hex(b1[i % len(b1)]), hex((i2 +
i-i1) % len(b2)), hex(b2[(i2 + i-i1) % len(b2)]), hex(b1[i % len(b1)] ^
b2[(i2 + i-i1) % len(b2)])) res[i-i1] = b1[i % len(b1)] ^ b2[(i2 + i-i1) %
len(b2)] return bytes(res) def find_stage2(data, key): sign =
b"\x11\x90\xea\xe9" #Alternative signature: b"\xe9\x4d\x01\x00" sign_off = 5
#Alternative offset: 0 dec = b"" for i in range(0, len(data), 4): dec = (dec
+ xor(data, i, key, 0, 4))[-8:] if sign in dec: j = dec.index(sign) return
i+j-4-sign_off def shift(data, val): return data[val:] + data[:val] def
decrypt_stage2(data): for k, v in count_dwords(data).items(): print(f"🤞
Possible (rotated) decrypt key: {hex(k)}") key = struct.pack("<I", k) offset
= find_stage2(data, key) if offset is None: print(f"😐 No stage found for
this key, trying next one.") continue else: print(f"Stage 2 found at offset
{hex(offset)}") key = shift(key, offset & 0x3) print(f"Adjusted key to:
{hex(struct.unpack('<I', key)[0])}") dec_data = data[:offset] + xor(data,
offset, key, 0, len(data)-offset) print(f"  Stage 2 decrypted.") return
dec_data def get_string_key(data): call_strdec = b"\xe8\xd9\xfe\xff\xff"
#The string key are the 0x2b bytes after this call. #If this fails, we can
try looking for all E8 (relative) calls if call_strdec in data: i =
data.index(call_strdec)+5 return data[i:i+0x2b] def find_strs(data, skey,
mlen=100): strs = [] i = 0 while i < len(data): possible = xor(data, i,
skey, 0, mlen) s = b""; for j in range(len(possible)): if (possible[j] >=
0x20 and possible[j] <= 0x7f) or possible[j] in [0xa, 0xd, 0x00, 0x07]: s +=
bytes([possible[j]]) else: break s2 = s.replace(b"\x00", b"") if len(s2) >=
5: strs.append(s2) i += len(s) i += 1 return strs def interesting_str(strs):
res = False for s in strs: if b"http" in s or b"://" in s: print(s) res =
True return res def extract_info(data): dec_data = decrypt_stage2(data)
print("Looking for the string key.") str_key = get_string_key(dec_data) if
str_key is None: print("💔 No string key found. Aborted.") return False
else: print(f"🥳 String key found: {hexlify(str_key)}") print("Finding
strings by bruteforce...") strs = find_strs(dec_data, str_key)
print("Interesting strings found:") return interesting_str(strs) # # MAIN #
if len(sys.argv) != 2: print(f"Usage: {sys.argv[0]} SHELLCODE_FILENAME",
file=sys.stderr) sys.exit(1) sys.exit(2 if not
extract_info(read_shellcode(sys.argv[1])) else 0)
```

Esempio

```
$ python3 gl.py ~/shared/guloader_shellcode
✌ Possible (rotated) decrypt key: 0x33e7b49b
Stage 2 found at offset 0x20ea
Adjusted key to: 0xb49b33e7
  Stage 2 decrypted.
Looking for the string key.
🥳 String key found:
b'0fc5fc4b7eb350b07d046090e4a0b73cb0100ed1063f658e0d43f257ec17708039314398012d065c9e46

Finding strings by bruteforce...
Interesting strings found:
b'https://91news.in/bcwq_WFnUhj158.bin'
```

Ottenuto il drop url è quindi possibile scaricare il payload. I primi 64 byte sono random e non usati, i restanti sono un PE xorato con una chiave.

Non essendo il payload disponibile al momento di questa analisi non abbiamo potuto automatizzare la sua decodifica. Si suggeriscono comunque due approcci:

- La chiave è solitamente tra i 0x200 e i 0x380 byte, i PE contengono spesso lunghe sequenze di byte nulli che rileverebbero la chiave. Cercando una sequenza ripetuta è possibile estrarre la chiave.
- Alcuni campi di un PE sono noti, questo rileva parte della chiave.

Aggiornamento

In seguito all'analisi di ulteriori sample è stato notato che la variabilità tra questi è troppo alta affinchè un approccio basato sul riconoscimento di firme (come avviene nello script sopra) possa funzionare.

L'alternativa è quella di utilizzare un approccio puramente bruteforce:

1. **Enumerare le prime n DWORD più presenti nello shellcode.** La speranza è che qualcuna di queste sia la chiave XOR per decodificare il secondo stadio (l'idea è che il secondo stadio contenga un numero elevato di zeri e quindi una volta cifrato un numero elevato di DWORD che corrispondono alla chiave).
2. **Per ogni DWORD, usarla come chiave per decodificare il secondo stadio.** Contrariamente a prima non sono fatti controlli riguardo la validità del secondo stadio ottenuto.
3. **Cercare tutte le chiamate dirette relative all'indietro, il cui offset sia compreso tra valori negativi piccoli (di default lo script usa -500 e -100)**. Questo passo identifica ogni possibile chiamata che delimita la chiave per decifrare le stringhe. Contrariamente a prima non sono fatte verifiche e tutti i candidati sono presi in considerazione.
4. **Usare tutte le chiavi candidate ottenute al punto 3 per decifrare le stringhe** e mostrare quelle che contengono determinati caratteri (es: http).

Oltre a questo approccio puramente bruteforce, è stata aggiunta la possibilità di continuare la ricerca quando viene trovata una stringa di interesse e soprattutto di salvare su file il secondo stadio decodificato. Questo tornerà utile per decodificare il payload.

```
import struct from binascii import hexlify import sys #After this many bytes, stop decoding a (so far valid) string. NOTE: Guloader uses UTF-16, the #no. of chars is halved! MAX_STRING_LEN = 150 #Try this many possible Stage2 decode keys MAX_STAGE2_DECODE_KEYS = 10 #The call before the String key must have an immediate between min and max (a bigger gap find more candidates) DECODE_STRING_MIN_IMMEDIATE = -500 DECODE_STRING_MAX_IMMEDIATE = -100 #Read a binary file def read_shellcode(filename): with open(filename, "rb") as f: data = f.read() return data #Count the DWORD in an array of bytes, not counting DWORD with the MSB equal to zero def count_dwords(data, skew=0): hist = {} for i in range(0 + skew, (len(data)-skew)//4 * 4, 4): dw = struct.unpack("<I", data[i:i+4])[0] if dw <= 0xffffff: continue if dw not in hist: hist[dw] = 1 else: hist[dw] += 1 return {x[0]:x[1] for i, x in enumerate(sorted(hist.items(), key=lambda x: -x[1])) if i < MAX_STAGE2_DECODE_KEYS} #XOR b1[i1, i1+l] with b2[i2:i2+l] and return the result (which has length l!) def xor(b1, i1, b2, i2, l): res = [0] * l for i in range(i1, i1+l): res[i-i1] = b1[i % len(b1)] ^ b2[(i2 + i-i1) % len(b2)] return bytes(res) #Find all calls with negative offset, not too big nor too small def get_string_keys(data): keys = [] for i in range(0, len(data)-4-0x2b): if data[i] != 0xe8: continue imm = struct.unpack("<i", data[i+1:i+5])[0] if imm >= DECODE_STRING_MAX_IMMEDIATE or imm < DECODE_STRING_MIN_IMMEDIATE: continue keys.append(data[i+5:i+5+0x2b]) return keys #Decode Strings def find_strs(data, skey, mlen=MAX_STRING_LEN): strs = [] i = 0 while i < len(data): possible = xor(data, i, skey, 0, mlen) s = b""; for j in range(len(possible)): if (possible[j] >= 0x20 and possible[j] <= 0x7f) or possible[j] in [0xa, 0xd, 0x00, 0x07]: s += bytes([possible[j]]) else: break s2 = s.replace(b"\x00", b"") if len(s2) >= 5: strs.append(s2) i += len(s) i += 1 return strs def interesting_str(strs): res = False for s in strs: if b"http" in s or b"://" in s: print(s) res = True return res def bruteforce(data): for k, v in count_dwords(data).items(): print(f"🤞 Possible (rotated) decrypt key: {hex(k)}") key = struct.pack("<I", k) print(f"👽 Decoding the shellcode...") dec_data = xor(data, 0, key, 0, len(data)) print(f"🔍 Finding the possible string keys...") keys = get_string_keys(dec_data) print(f"🏋 Found {len(keys)} keys. Brute forcing...") for k in keys: print(f"🤞 Trying key {hexlify(k)}") strs = find_strs(dec_data, k) if interesting_str(strs): while True: action = input("Type c to continue the search, q to quit, s FILENAME to save the decoded stage and exit: ").split(" ", 2) cmd = action[0].lower() if cmd == "q": return True elif cmd == "c": break elif cmd == "s": with open(action[1].strip(), "wb") as f: f.write(dec_data) return True return False # # MAIN # if len(sys.argv) != 2: print(f"Usage: {sys.argv[0]} SHELLCODE_FILENAME", file=sys.stderr) sys.exit(1) sys.exit(2 if not bruteforce(read_shellcode(sys.argv[1])) else 0)
```

Il payload scaricabile dal drop URL è codificato tramite XOR con una chiave la cui lunghezza (e valore) cambia da sample a sample.

La chiave non è salvata nel secondo stadio in chiaro ma è a sua volta XORata con una WORD (intero di 16 bit) che guloader calcola a runtime tramite bruteforcing. L'algoritmo che usa Guloader è il seguente.

```
//Chiave (sconosciuta per noi)
uint16_t key = { ... };
//Payload scaricato
uint16_t* payload = ...;

//Calcolo della WORD da xorare con la chiave
uint16_t index;
for (index = 0; (uint32_t)index < 0x10000; index++)
   if (key[0] ^ index ^ payload[32] == 0x4d5a)
      break;

//Calcolo della chiave
for (int j = 0; j < sizeof(key)/sizeof(key[0]); j++)
   key[j] ^= index;
```

Guloader può calcolare *index* perchè sà che la prima WORD di un PE è `0x4d5a` (MZ) e perchè conosce dove è la chiave codificata e la sua lunghezza.

Noi **non** conosciamo dove si trova la chiave **nè** la sua lunghezza ma possiamo di nuovo sfruttare della Crittoanalisi 101 per montare un attacco bruteforce.

L'idea è che conosciamo i primi due byte della chiave grazie alla firma MZ (chiamiamoli *k0*), k0può comparire all'interno del PE sia perchè la WORD corrispondente nel PE era zero (0 ^ *k0* = *k0*) sia perchè per coincidenza l'operazione di XOR l'ha data come risultato (*word_nel_pe_originale* ^ *ki* = *k0*).

Supponiamo che troviamo solo instanze del primo tipo, la distanza in byte tra due di queste WORD di valore *k0* è un multiplo della chiave. E' un multiplo e non la lunghezza esatta perchè non possiamo garantire che tutte le istanze di k0 compaiano nel PE codificato. Per cui prendendo la più piccola lunghezza abbiamo buona probabilità di trovare la lunghezza effettiva della chiave.
Istanze del secondo tipo (che, con un po' di forzatura, assunto una distribuzione uniforme di chiave e payload si verificano con probabilità 2^-16) possono indurre falsi positivi. Per cui si ottiene una lista di possibili lunghezze (tutte quelle che non sono multiple di altre).

Alla fine si ha un insieme di possibili lunghezze di chiavi, se si è fortunati se ne ha una solo. Se le lunghezze ottenute sono tutte multiple della lunghezza effettiva l'attacco sotto fallisce, si può pensare in questo caso di fattorizzare le lunghezze trovare e considerare tutti i possibili divisori (questo è ancora da implementare).

Per trovare la chiave stessa verrebbe da provare a cercare una sequenza di byte che inizia con *k0* e che si ripete (almeno in parte). Ma le chiavi usate possono essere troppo lunghe perchè questo succeda, ad esempio nel sample usato la chiave era di `0x606` byte, troppo lunga affinchè il PE avesse tutti questi zeri consecutivi.

Un altro approccio è utilizzare il secondo stadio decodificato e tentare un bruteforce.
Scorriamo ogni singola WORD *s0* nel secondo stadio e la consideriamo come l'inizio della chiave, che ricordiamo è XORata con la quantità *index*. Dato che conosciamo per certo *k0*, possiamo calcolare *index* = *s0* ^ *k0* visto che *s0* = *k0* ^ *index*. Se davvero *s0* è l'inizio della chiave codificata nel secondo stadio, calcolando *si* ^ *ki*, dove *si* sono le WORD successive a *s0* nel secondo stagio e *ki* quelle successive a *k0* nel payload cifrato, per ogni *i* fino a raggiungere la lunghezza stimata, la maggior parte di questi valori sarà pari ad *index*.
Non tutti saranno uguale ad *index* perchè i *ki* sono in realtà WORD che vengono dal payload codificato e corrispondono alla WORD *ki* chiave solo se e solo se in quella posizione il PE conteneva una WORD nulla.
Tuttavia prendendo come candidati le chiavi che danno almeno *m* valori uguali ad *index* (di default *m*=10) si ha una buona probabilità di trovare dove inizia la chiave nel secondo stadio e il valore *index*.

Trovati *index*, la chiave nel secondo stadio e la sua lunghezza, è possibile emulare la decodifica di Guloader ed ottenere il payload.

Lo script seguente prende da linea di comando il percorso del secondo stadio decodificato (generato dallo script sopra ad esempio) e del payload cifrato (così come scaricato) e prova un attacco bruteforce per ottenere il payload.

```
 import binascii import struct import pefile import sys #Xor b1[i1:i1+l]
with b2[i2:i2+l] and return a byte array of length l def xor(b1, i1, b2, i2,
l): res = [0] * l for i in range(i1, i1+l): res[i-i1] = b1[i % len(b1)] ^
b2[(i2 + i-i1) % len(b2)] return bytes(res) #Find the possible lengths of
the key and the possible keys (only those with a "primitive" length will be
effetively used) def find_keys_and_lens(data, min_len=0x100): #We know the
first two bytes of the PE, so we know the first two bytes of the key
key_start = xor(b"MZ", 0, data, 0, 2) print(f"Key start is
{binascii.hexlify(key_start)}") #Where was the last WORD with value
key_start last_start = None #The keys found keys = [] #The lengths found (#
of these is <= # keys as two or more keys can share a length) lens = []
#Scan all the payload WORDs for i in range(0, len(data), 2): #If not a key
start, skip if data[i:i+2] != key_start: continue #If this is the second key
start, save the key and the length if not already present if last_start is
not None : pkey = data[last_start:i] if pkey not in keys: print(f"Found a
possible key at offset {hex(i)} with (possible multiple) len
{hex(len(pkey))}") keys.append(pkey) last_start = i #We remove all the
length that are multiple of other lengths or too low for k in sorted(keys,
key = lambda k: len(k)): l = len(k) if l < min_len: continue #First (and
```

```
smallest) length if len(lens) == 0: lens.append(l) #No multiples? Add elif
len([ol for ol in lens if l % ol == 0]) == 0: lens.append(l) return keys,
lens # # M A I N # if len(sys.argv) != 3: print(f"Usage: {sys.argv[0]}
DECODED_STAGE2_FILENAME PAYLOAD_FILENAME") sys.exit(1) #Read the data with
open(sys.argv[2], "rb") as f: data = f.read()[64:] with open(sys.argv[1],
"rb") as f: stage2 = f.read() #Get keys and lengths keys, lens =
find_keys_and_lens(data) print(f"Found {len(lens)} possible key length(s)")
#TODO: Show the key lengths and ask if we should add each divisor (if
greater than a threshold) of these length to the list (and the relative key
prefixes to keys) before bruteforcing. # if the script fails to find the
payload, try implementing this, even manually. #For each key length... for
kl in lens: #Get the possibly partially coded keys from the payload
candidates = [k for k in keys if len(k) == kl] print(f"Trying keys with len
{hex(kl)} ({len(candidates)} candidate(s) found)") #For each candidate n = 0
for c in candidates: #This WORD is known to be the valid (it's the first
WORD of the key) k0 = struct.unpack("<H", c[0:2])[0] print("Looking for a
match in the decoded stage2...") #For each WORD in the second stage... for i
in range(0, len(stage2)-kl): #Progress if i % 10000 == 0: print(f"Still
looking... ({i*100//(len(stage2)-kl)}% of stage2 checked)") #Calculate the
possible index s0 = struct.unpack("<H", stage2[i:i+2])[0] index = s0 ^ k0
#Count how many times index comes up when decodind the subsequent words
count_matches = 0 for j in range(2, kl, 2): si = struct.unpack("<H",
stage2[i+j:i+j+2])[0] ki = struct.unpack("<H", c[j:j+2])[0] if si ^ ki ==
index: count_matches += 1 #If we have at least 10 matches, consider this a
possible key if count_matches >= 10: print(f"A candidate matched") #Find the
key key = xor(stage2, i, struct.pack("<H", index), 2, kl) #Decode the PE pe
= xor(data, 0, key, 0, len(data)) #Try parsing the PE try:
pefile.PE(data=pe) name = f"Payload{n}.exe" print(f"Possible key found,
saving payload to {name}") with open(name, "wb") as f: f.write(pe) except
Exception as e: continue
```

```
$ python3 gl3.py gu_s2.bin ~/Malwares/20220727/gumabelt_DNCAoUwjFj89.bin
Key start is b'5848'
Found a possible key at offset 0x5c3a with (possible multiple) len 0x3d7c
Found a possible key at offset 0x10862 with (possible multiple) len 0xac28
Found a possible key at offset 0x114ae with (possible multiple) len 0xc4c
Found a possible key at offset 0x120fa with (possible multiple) len 0xc4c
Found a possible key at offset 0x12d46 with (possible multiple) len 0xc4c
Found a possible key at offset 0x1522a with (possible multiple) len 0x24e4
Found a possible key at offset 0x1770e with (possible multiple) len 0x24e4
Found a possible key at offset 0x20a9e with (possible multiple) len 0x9390
Found a possible key at offset 0x28bbc with (possible multiple) len 0x811e
Found a possible key at offset 0x2b6c6 with (possible multiple) len 0x2b0a
Found a possible key at offset 0x2c312 with (possible multiple) len 0xc4c
Found a possible key at offset 0x33e0a with (possible multiple) len 0x7af8
Found a possible key at offset 0x34430 with (possible multiple) len 0x626
Found 1 possible key lengths
Trying keys with len 0x626 (1 candidate(s) found)
Looking for a match in the decoded stage2...
Still looking... (0% of stage2 checked)
Still looking... (11% of stage2 checked)
Still looking... (22% of stage2 checked)
Still looking... (33% of stage2 checked)
Still looking... (44% of stage2 checked)
Still looking... (55% of stage2 checked)
Still looking... (66% of stage2 checked)
A candidate matched
Possible key found, saving payload to Payload0.exe
Still looking... (78% of stage2 checked)
Still looking... (89% of stage2 checked)
$ file Payload0.exe
Payload0.exe: PE32 executable (GUI) Intel 80386 Mono/.Net assembly, for MS Windows
```

Aggiornamento

E’ disponibile anche la versione C degli stessi script. I parametri di ricerca sono definiti in config.h.

L’utilizzo è il seguente :

```
gudecoder url FILE_STAGE2_CIFRATO
...
gudecoder payload FILE_STAGE2_DECIFRATO PAYLOAD_CIFRATO
```

Taggato guloader