Payload Threat Actor Ransomware

Published: 2026-04-05 · Archived: 2026-04-10 02:07:48 UTC

Payload Threat Actor

The current information about this new threat actor in the wild till this moment is little , but according to Ahmed Elessaway , It is active since **17-02-2026** and till this moment has targets between `United States` , `Philippines` , `Mexico` , `United Kingdom` and `Egypt` , and it reached 26 victims.

▶ Victims

Sample information

Field	Value
File name	`locker_esxi.elf`
File format	`ELF64 Linux x86-64`
File size	`0x9BE0` (stripped on disk, `0x2097A8` mapped)
MD5	`f91cbdd91e2daab31b715ce3501f5ea0`
SHA1	`0252819a4960c56c28b3f3b27bf91218ffed223a`
SHA256	`bed8d1752a12e5681412efbb8283910857f7c5c431c2d73f9bbc5b379047a316`

Executive Summary

`locker_esxi.elf` is a 64-bit Linux ELF ransomware binary targeting **VMware ESXi hypervisor environments** . The sample combines a robust cryptographic scheme `Curve25519 ECDH` and `ChaCha20` with ESXi-specific VM enumeration via the **vmInventory.xml** inventory file, graceful shutdown of running VMs before encryption, and a **multi-threaded file encryption** pipeline scaled to available CPU cores. The ransom note is delivered inside ESXi's own web UI `welcome.txt` , replacing the host management interface greeting.

String Decryption

The binary embeds its configuration as `RC4-encrypted` and `base64-encoded` blobs in the `.rodata` section. The RC4 key is the three-byte **FBI**. All sensitive **strings** , **file paths** , **error messages** and **shell commands** are decrypted at runtime through `mw_w_RC4()` .

```c
char *__fastcall mw_w_RC4(__int64 input, __int64 len)
{
  _BYTE key[282]; // [rsp+Eh] [rbp-11Ah] BYREF

  mw_RC4_KSA(key, "FBIthread-pool-%d", 3uLL);
  __memcpy_chk(byte_609460, input, len, 0x100LL);
  mw_RC4_PRGA(key, byte_609460, len);
  byte_609460[len] = 0;
  return byte_609460;
}
```

Figure(1) mw_w_RC4() Decryption Function

The python decryption script below to decrypt data blobs in place in IDA to make the analysis more easier.

```python
import idc
import idaapi

def rc4(key: bytes, data: bytes) -> bytes:
    S = list(range(256))
    j = 0
    for i in range(256):
        j = (j + S[i] + key[i % len(key)]) % 256
        S[i], S[j] = S[j], S[i]

    result = bytearray()
    i = j = 0
    for byte in data:
        i = (i + 1) % 256
        j = (j + S[i]) % 256
        S[i], S[j] = S[j], S[i]
        result.append(byte ^ S[(S[i] + S[j]) % 256])
    return bytes(result)


RC4_KEY = b"FBI"

BLOBS = [
    (0x6093F0, 0x12),
    (0x6092A0,   33),
    (0x609288,   13),
    (0x609260,   14),
    (0x60925A,    5),
    (0x609250,   10),
    (0x609270,   18),
    (0x609300,   34),
    (0x609380,   43),
    (0x609330,   22),
    (0x609350,   28),
    (0x609220,   44),
    (0x6092D0,   30),
    (0x6093D0,   31),
    (0x6093B0,   20),
```

```
39  ]
40
41
42  def patch_blob(ea: int, plaintext: bytes) -> None:
43      for offset, byte in enumerate(plaintext):
44          idc.patch_byte(ea + offset, byte)
45      idc.patch_byte(ea + len(plaintext), 0)
46
47
48  def make_string(ea: int, length: int) -> None:
49      idaapi.del_items(ea, idaapi.DELIT_SIMPLE, length + 1)
50      idc.create_strlit(ea, ea + length + 1)
51
52
53  def main():
54      for ea, size in BLOBS:
55          data = idc.get_bytes(ea, size)
56          if data is None:
57              continue
58          plaintext = rc4(RC4_KEY, data)
59          patch_blob(ea, plaintext)
60          make_string(ea, size)
61
62
63  if __name__ == "__main__":
64      main()
```

Decrypted strings:

▶ Show List

Technical Analysis

Anti-Analysis & Defense Evasion

Malware immediately at beginning checks if debugger attached to the process by opening **/proc/self/status** and reads lines until it finds `TracerPid:`, if the integer following that field is non-zero so debugger is attached , execution branches to delete itself.

```c
_BOOL8 mw_check_debugger()
{
  FILE *v0; // rbx
  _BOOL8 result; // rax
  int v2; // ebp
  char v3[10]; // [rsp+0h] [rbp-118h] BYREF
  char nptr[270]; // [rsp+Ah] [rbp-10Eh] BYREF

  v0 = fopen("/proc/self/status", "r");
  result = 0LL;
  if ( v0 )
  {
    while ( fgets(v3, 256, v0) )                // opens /proc/self/status and reads lines until it finds TracerPid:
    {
      if ( !strncmp(v3, "TracerPid:", 0xAuLL) )
      {
        v2 = atoi(nptr);                        // If the integer following that field is non-zero a debugger is attached
        fclose(v0);
        return v2 != 0;
      }
    }
    fclose(v0);
    return 0LL;
  }
  return result;
}
```

Figure(2) Malware checking for debugger

Cryptographic Scheme

The cryptographic scheme takes stages before doing the actual encryption such as **encryption public key extraction** , **gathering CPU capabilities** , **encryption routine selection** , **thread-pool initialization** and finally **multi-thread encryption job**.

The malware encodes its public key via **Base-64** encoding shown in figure below , which reveling the public encryption key is `3E67A9F94526785C31C543BE3F4DC7039E7C3F764F65637C6C22B85F3357B575` .

```c
__int64 mw_decode_pub_key()
{
  _BYTE *decoded_pub_key; // rax
  char *v1; // rax
  void *v2; // rbx
  char *v3; // rax
  __int64 pub_key_size; // [rsp+8h] [rbp-10h] BYREF

  pub_key_size = 0LL;
  decoded_pub_key = mw_base64_decode(off_609428, &pub_key_size);// Pmep+UUmeFwxxUO+P03HA558P3ZPZWN8bCK4XzNXtXU=
  if ( !decoded_pub_key )
  {
    v1 = mw_w_RC4(aBase64PubkeyDe, 0x1FLL);     // [!] base64 pubkey decode failed
    puts(v1);
    return 0xFFFFFFFFLL;
  }
  v2 = decoded_pub_key;
  if ( pub_key_size != 32 )
  {
    v3 = mw_w_RC4(aInvalidKeySize, 0x14LL);     // [!] invalid key size
    puts(v3);
    free(v2);
    return 0xFFFFFFFFLL;
  }
  pub_key = decoded_pub_key;                    // 3E67A9F94526785C31C543BE3F4DC7039E7C3F764F65637C6C22B85F3357B575
  return 0LL;
}
```

Figure(3) Public Key Extraction

Gathering CPU Capabilities

The malware wants to run **ChaCha20** as fast as possible, so before any encryption happens it probes the CPU for SIMD capabilities and stores a function pointer to the fastest available implementation. Every subsequent file encryption call goes through that pointer without re-checking.

```
  _RAX = 1LL;
  __asm { cpuid }                               // CPUID with EAX=1 returns feature flags in EDX and ECX
  v5 = (_RDX >> 26) & 1;                        // // bit 26 of EDX = SSE2 present?
  v6 = v5;
  if ( (_RCX & 0x10000000) == 0 )
    goto LABEL_5;
  if ( (_RCX & 0x8000000) == 0 )                // bit 27 ECX = AVX present?
    goto LABEL_5;
  __asm { xgetbv }
  v7 = v5 & 6;
  v8 = v7 == 6;
  if ( v7 == 6 )
    v6 |= 2u;
  else
LABEL_5:
    v8 = 0;
  _RAX = 7LL;
  __asm { cpuid }
  if ( v8 && (_RBX & 0x20) != 0 )               //  AVX usable AND bit 5 of EBX = AVX2
    v6 |= 4u;
  return v6;
}
```

Figure(4) Gathering CPU Capabilities The function queries processor feature flags via `CPUID` and `XGETBV` :

- Checks **SSE2 support** via `EDX bit 26` .
- Checks **AVX support** via `ECX bits 28/27` .
- Uses `XGETBV` to confirm OS-level AVX state (XSAVE enabled state validation).
- Queries extended features `CPUID leaf 7` to detect **AVX2 support (EBX bit 5)**.

The resulting capability mask is built as:

- `bit 0` **value 0** so SSE2 available.
- `bit 1` **value 1** so AVX usable.
- `bit 2` **value 4** so AVX2 available. This behavior is consistent with performance-aware malware or protected loaders that adapt cryptographic implementations based on host hardware capabilities.

Encryption Routine Variants

Based on the CPU capabilities it decides between which routine will handle the encryption logic.

```
cpu_flags = mw_get_CPU_capabilities();
if ( (cpu_flags & 4) != 0 )
{
  ptr_encryption_engine = mw_chacha20_avx2;
}
else
{
  v6 = (cpu_flags & 1) == 0;
  v7 = mw_chacha20_scalar;
  if ( !v6 )
    v7 = mw_chacha20_sse2;
  ptr_encryption_engine = v7;
}
```

Figure(5) Encryption routine selection

mw_chacha20_scalar

- **Pure scalar implementation** using 32-bit general-purpose registers (`EAX` , `EBX` , `ECX` , etc.)
- Entire ChaCha20 state maintained as **individual integers** (heavily stack-spilled).
- Rotations implemented manually via `__ROL4__` (shift + OR).
- Processes **1 block (64 bytes) per iteration**.
- Block counter increments by **+1**.
- Simple control flow:
  - Single loop of **10 double-rounds**.

  - Followed by a separate **add-back phase**.
- XOR with plaintext performed **4 bytes at a time**.
- Most **portable but least performant** variant.

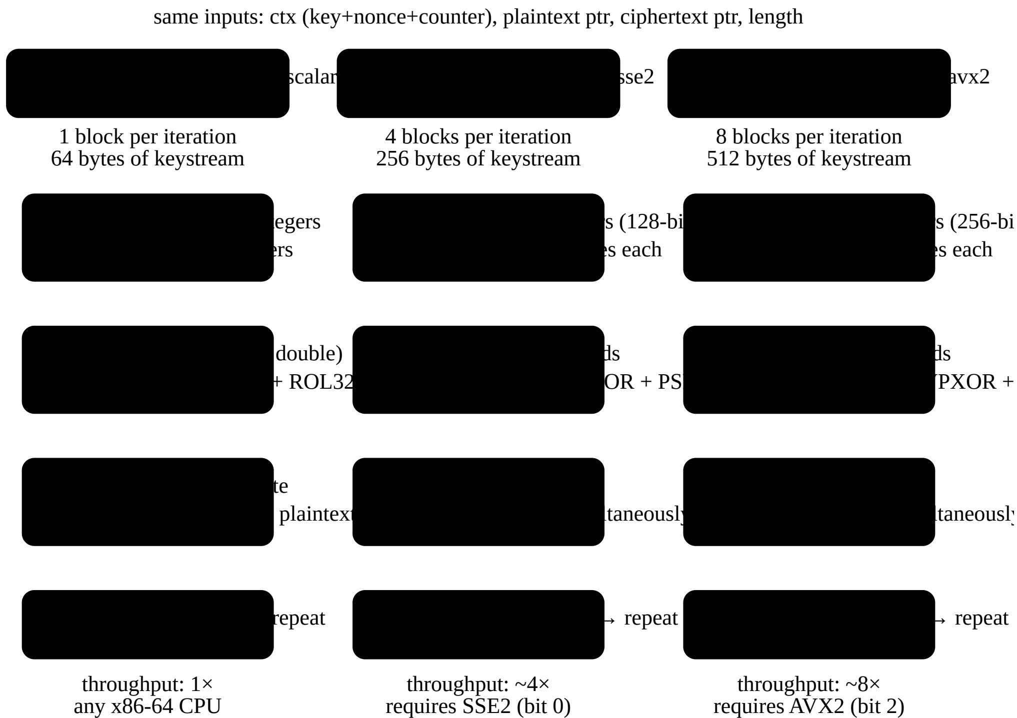
mw_chacha20_sse2

- Uses **128-bit XMM registers** with VEX-encoded SSE instructions: `vpaddd` , `vpxor` , `vpshufd` , `vpslld` , `vpsrld` .
- Implements **4-way parallelism** each XMM lane represents one block and processes **4 blocks simultaneously**.
- Counter initialization: uses `vpaddq` with precomputed constants to generate **4 parallel counters**.
- Block counter increments by **+4 per iteration**.
- Rotation strategy:
  - Bit rotations via shift+OR.
  - Lane permutations via `vpshufd` (shuffles `0x93` , `0x4E` , `0x39`).
- Output: XOR and writes using XMM registers **16 bytes per store**.
- Loop structure:
  - Outer loop **256-byte aligned chunks (4×64)**.
  - Tail loop remaining 64-byte blocks.
  - Sub-64-byte remainder handled via **stack buffer staging**.

mw_chacha20_avx2

- Fully utilizes **256-bit YMM registers**.
- Implements **8-way parallelism** each YMM lane holds one word across **8 blocks**.
- Setup phase: `vbroadcasti128` used to duplicate key/state across YMM lanes.
- Advanced data movement: `vperm2i128` enables **cross-lane shuffling** (critical for 8-block transposition).
- Block counter increments by **+8 per iteration**.
- Rotation optimization: uses `vpshufb` with precomputed masks which **faster than shift+OR**.
- Output: writes **32 bytes per store** (double SSE2 throughput).
- Loop hierarchy:
  - Main loop **512-byte aligned chunks (8×64)**
  - Secondary loop **128-byte tail**
  - Final stage **fine-grained remainder handling** (branching per 32-byte boundary)

And the figure below explains the stages of every routine



Figure(6) ChaCha20 Three Variants

Thread Pool Execution

The sample initializes a multi-threaded processing pipeline by creating a **thread pool sized at 2× the number of CPU cores**, indicating intent for high-throughput file operations.

It accepts an optional command-line exclusion list `(-i)`, allowing specific ports to be skipped during execution suggesting operator-controlled targeting or selective encryption.

The malware then accesses a VMware ESXi configuration file `(/etc/vmware/hostd/vmInventory.xml)`, with both the file path and `XPath query (//configentry)` RC4-obfuscated , It parses <configentry> nodes to extract **port identifiers** (used as directory references) **datastore paths**(VM storage locations).

The malware then iterates over collected directories and:

- Enumerates files within each directory.
- Skips files already marked as encrypted (**suffix .xx0001**).

- Submits remaining files as jobs to the thread pool for encryption via `mw_encrypt_file`.

```
v1 = opendir(a1);
v2 = v1;
if ( v1 )
{
  while ( 1 )
  {
    v3 = readdir64(v2);
    if ( !v3 )
      break;
    d_name = v3->d_name;
    if ( strcmp(v3->d_name, ".") )
    {
      if ( strcmp(d_name, "..") )
      {
        __snprintf_chk(s, 4096LL, 1LL, 4096LL, "%s/%s", a1);
        if ( sub_406370(s, &stat_buf) != -1 && (stat_buf.st_mode & 0xF000) == 0x8000 )
        {
          if ( !off_609410
            || (v5 = strlen(d_name) + 1, v6 = strlen(off_609410) + 1, v5 - 1 < v6 - 1)
            || strcmp(&d_name[v5 - v6], off_609410) )// offset_609410 = .xx0001
          {
            v7 = strdup(s);
            mw_threadpool_add_job(ptr, mw_encrypt_file, v7);
          }
        }
```

Figure(7) Malware encrypts files and suffix .xx0001

Each worker thread on startup calls `prctl(PR_SET_NAME, "FBIthread-pool-%d")` — the thread name string `FBIthread-pool-%d` is embedded in plaintext and is a notable forensic indicator. On wakeup, a thread dequeues a job, executes it as `job->fn(job->arg)`, frees the job struct, and decrements the active-job counter. If the active count drops to zero, a `pthread_cond_signal` is fired to unblock any waiting caller of threadpool wait.

Encryption Routine

The function implements a targeted, in-place ransomware encryption workflow optimized for `ESXi/VMware environments`. It exclusively processes files larger than **5 GB**, effectively filtering for virtual disk images such as **VMDK** while ignoring smaller files, indicating deliberate high-value targeting.

Each file undergoes a per-file `Curve25519 ECDH key exchange`, generating a unique ephemeral key pair and deriving a ChaCha20 encryption key from a decoded attacker public key. This ensures strong cryptographic isolation, preventing decryption without attacker-controlled material.

Encryption is performed partially across 5 segments, each up to **1 GB**, using a 1 MB sliding buffer and executed in-place (no temporary files). The ChaCha20 keystream is synchronized with file offsets, enabling deterministic decryption while significantly reducing processing time—typical optimizing for speed on large datasets.

```
{
  cur_file_offset = segment_offset;
  chunk_end_offset = segment_offset + 0x40000000;
  LOWORD(chacha_counter_lo) = segment_offset >> 6;
  HIBYTE(chacha_counter_lo) = (segment_offset >> 6) >> 0x18;
  BYTE2(chacha_counter_lo) = (segment_offset >> 6) >> 0x10;
  sub_402F7F(chacha_ctx, &chacha_counter_lo);
  while ( cur_file_offset < chunk_end_offset )
  {
    fseeko64(fp, cur_file_offset, 0);
    read_size = chunk_end_offset - cur_file_offset;
    if ( chunk_end_offset - cur_file_offset > 0x100000 )
      read_size = 0x100000LL;
    bytes_read = __fread_chk(io_buf, 0x100000LL, 1LL, read_size, fp);
    bytes_read_sz = bytes_read;
    if ( !bytes_read )
      break;
    ptr_encryption_engine(chacha_ctx, io_buf, io_buf, bytes_read);
    fseeko64(fp, cur_file_offset, 0);
    cur_file_offset += bytes_read_sz;
    fwrite(io_buf, 1uLL, bytes_read_sz, fp);
  }
  segment_size_this = base_segment_size + (remainder_segments > segment_idx++);
  segment_offset += segment_size_this;
}
```

Figure(8) Encryption routine

Post-encryption, the malware appends a **56-byte footer** containing the ephemeral public key, lightly obfuscated using RC4 3-byte key `("FBI")`. This serves as metadata for decryption while providing minimal resistance to static extraction. Sensitive key material is explicitly wiped from memory, reflecting anti-forensic intent.

```
while ( segment_idx != 5 );
curve25519_basepoint[2] = 0x64616F6C796170LL;
free(io_buf);
mw_RC4_KSA(chacha_ctx, "FBIthread-pool-%d", 3uLL);
mw_RC4_PRGA(chacha_ctx, ephem_pubkey_buf, 0x38LL);
fseeko64(fp, 0LL, 2);
fwrite(ephem_pubkey_buf, 1uLL, 0x38uLL, fp);
footer_wipe_ptr = ephem_pubkey_buf;
for ( j = 0xELL; j; --j )
  *footer_wipe_ptr++ = 0;
encrypt_success = 1;
}
```

Figure(9) Metadata For Decryption

Ransom Note

The ransom note inside **ESXi's** own web UI `welcome.txt`, replacing the host management interface greeting.

```
1   Welcome to Payload!
2
3   The next 72 hours will determine certain factors in the life of your company:
4   the publication of the file tree, which we have done safely and unnoticed by all of you,
5   and the publication of your company's full name on our luxurious blog.
6   NONE of this will happen if you contact us within this time frame and our negotiations are favorable.
7
8   We are giving you 240 hours to:
9   1. familiarize yourself with our terms and conditions,
10  2. begin negotiations with us,
11  3. and successfully conclude them.
12  The timer may be extended if we deem it necessary (only in the upward direction).
13  Once the timer expires, all your information will be posted on our blog.
14
15  ATTENTION!
```

```
16  Contacting authorities, recovery agencies, etc. WILL NOT HELP YOU!
17  At best, you will waste your money and lose some of your files, which they will carefully take to restore!
18  You should also NOT turn off, restart, or put your computer to sleep.
19  In the future, such mistakes can make the situation more expensive and the files will not be restored!
20  We DO NOT recommend doing anything with the files, as this will make it difficult to recover them later!
21
22  When contacting us:
23  you can request up to 3 files from the file tree,
24  you can request up to 3 encrypted files up to 15 megabytes
25  so that we can decrypt them and you understand that we can do it.
26
27  First, you should install Tor Browser:
28  1. Open: https://www.torproject.org/download
29  2. Choose your OS and select it
30  3. Run installer
31  4. Enjoy!
32
33  In countries where tor is prohibited, we recommend using bridges,
34  which you can take: https://bridges.torproject.org/
35
36  You can read:
37  [NOPE 3eb] (Tor)
38
39  To start negotiations, go to [NOPE] and login:
40  User: [snip]
41  Password: [snip]
42
43  Your ID to verify: [snip]
```

MITRE ATT&CK Coverage

ID	Technique	Description
T1622	Debugger Evasion	TracerPid check + self-deletion
T1027	Obfuscated Files/Info	RC4 string obfuscation
T1036.005	Process Masquerading	prctl thread name spoofing
T1070.004	File Deletion	Self-deletes binary on detection
T1082	System Info Discovery	CPUID, sysconf
T1083	File and Directory Discovery	opendir / readdir64
T1486	Data Encrypted for Impact	ChaCha20
T1490	Inhibit System Recovery	Targets VMDK / datastore images
T1059.004	Unix Shell Execution	vim-cmd via system()

IOCs

- Files bearing `.xx0001` extension on ESXi datastores
- 56-byte footer containing hex magic `70 61 79 6C 6F 61 64 00`
- Processes named FBIthread-pool-0, FBIthread-pool-1, … visible in ps/top
- Ransom note present at `/usr/lib/vmware/hostd/docroot/ui/welcome.txt`

YARA Rule

```
1   rule payload_ransomware
2   {
3       meta:
4           description      = "Payload ESXi Ransomware"
5           author           = "Abdullah Islam @0x3oBAD"
6           date             = "2026-04-07"
7           md5              = "f91cbdd91e2daab31b715ce3501f5ea0"
8           sha256           = "bed8d1752a12e5681412efbb8283910857f7c5c431c2d73f9bbc5b379047a316"
9           malware_family   = "Payload Ransomware"
10          target_platform  = "Linux / VMware ESXi"
11          detection_phases = "pre-install dropper, execution, lateral-movement artifact"
12
13      strings:
14          $curve25519_clamp = { C0 ?? F8 ?? ?? ?? ?? 40 }
15          $chacha20_const   = "expand 32-byte k" ascii
16
17          $proc_status      = "/proc/self/status" ascii
18          $tracer_field     = "TracerPid:" ascii
19          $proc_exe         = "/proc/self/exe" ascii
20
21          $esxi_note        = "/usr/lib/vmware/hostd/docroot/ui/welcome.txt" ascii
22          $urandom          = "/dev/urandom" ascii
23
24          $vim_cmd          = "vim-cmd vmsvc/power.off %d > /dev/null 2>&1" ascii
25          $esxi_inventory   = "/etc/vmware/hostd/vmInventory.xml" ascii
26          $xpath_query      = "//ConfigEntry" ascii
27          $vmx_field        = "vmxCfgPath" ascii
28
29          $ext              = ".xx0001" ascii nocase
30          $footer_magic     = { 70 61 79 6C 6F 61 64 00 }
31
32          $thread_name      = "FBIthread-pool-%d" ascii
33          $thpool_init_oom  = "thpool_init(): Could not allocate memory for thread pool" ascii
34          $thpool_job_oom   = "thpool_add_work(): Could not allocate memory for new job" ascii
35          $thpool_sigusr    = "thread_do(): cannot handle SIGUSR1" ascii
36          $bsem_err         = "bsem_init(): Binary semaphore can take only values 1 or 0" ascii
37
38          $pubkey_1         = "TnJqU2F5RFFYREpPTURkUGx5Q3NMem0yNlZKM0s1aks=" ascii
39          $pubkey_2         = "Pmep+UUmeFwxxUO+P03HA558P3ZPZWN8bCK4XzNXtXU=" ascii
40          $pubkey_head      = "W2M6q8YwDKCcSvBj7YRjVNtSI/PO22G+" ascii
41
42          $key_err_1        = "[!] invalid key size" ascii
43          $key_err_2        = "[!] base64 pubkey decode failed" ascii
```

```
44
45            $libxml2          = "libxml2.so.2" ascii
46            $b64_alpha        = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/" ascii
47
48            $segment_loop_5   = { B? 05 00 00 00 [0-16] 3? 05 }
49            $flock_lockex     = { B? 02 00 00 00 [0-8] E8 ?? ?? ?? ?? }
50
51        condition:
52            uint32(0) == 0x464C457F
53            and uint8(4)  == 2
54            and uint8(5)  == 1
55            and uint16(0x12) == 0x3E
56
57            and (
58                (
59                    filesize == 39904
60                    and all of ($pubkey_1, $pubkey_2, $pubkey_head)
61                )
62                or (
63                    $proc_status
64                    and $tracer_field
65                    and $proc_exe
66                    and (
67                        any of ($pubkey_1, $pubkey_2, $pubkey_head)
68                        or ($key_err_1 and $key_err_2)
69                    )
70                    and 1 of ($chacha20_const, $curve25519_clamp, $b64_alpha)
71                )
72                or (
73                    $vim_cmd
74                    and $esxi_inventory
75                    and $xpath_query
76                    and (
77                        $esxi_note
78                        or ($thpool_init_oom and $thread_name)
79                        or ($flock_lockex and $ext)
80                    )
81                    and (
82                        #esxi_note + #vmx_field + #xpath_query + #ext
83                        + #thread_name + #thpool_init_oom + #thpool_job_oom
84                        + #thpool_sigusr + #bsem_err + #key_err_1
85                        + #key_err_2 + #proc_exe + #urandom + #libxml2 >= 5
86                    )
87                )
88                or (
89                    $chacha20_const
90                    and $curve25519_clamp
91                    and $urandom
92                    and 2 of ($vim_cmd, $esxi_inventory, $thread_name, $xpath_query)
93                )
94                or (
95                    $segment_loop_5
96                    and $flock_lockex
97                    and $footer_magic
98                    and 1 of ($esxi_inventory, $vim_cmd, $ext)
```

```
99              )
100             or (
101                 $footer_magic at (filesize - 56 + 24)
102             )
103         )
104 }
```

Conclusion

`locker_esxi.elf` is a targeted **ESXi ransomware** focused on encrypting large VM disk files **(>5 GB)** for maximum impact. It uses `Curve25519 + ChaCha20` with per-file keys, ensuring strong cryptographic isolation. The malware leverages multi-threading and SIMD optimizations to accelerate encryption across systems. It demonstrates environment awareness by parsing VMware configs and shutting down VMs before encryption. Overall, it is a highly efficient, enterprise-focused ransomware designed for rapid disruption of virtualized infrastructure.

Source: https://0x3obad.github.io/posts/payload-ransomware-writeup/