Intercepting Entropy – Hooking PRNG to Recover Ransomware Encryption Keys

Virus Bulletin 2025
Raviv Rachmiel

vb 2025 BERLIN 24-26 Sept 2025

WHO AM I

- Cybersecurity expert with 9 years of experience
- Software Engineer Alumni @ Technion, IL
- MBA Alumni @ TAU, IL + GWU, US
- Expertise in Malware Research and Reverse Engineering
- TA @ Technion, IL
- Head of a cybersecurity R&D group
- Completed the Flare-on challenge 3 times (hopefully this year will be the 4th)
- Cyber security Consulting Services – RE, MR, Cloud research and IR.
- Kitesurfer!

MY MISSION

- Finding a solution to ransomware

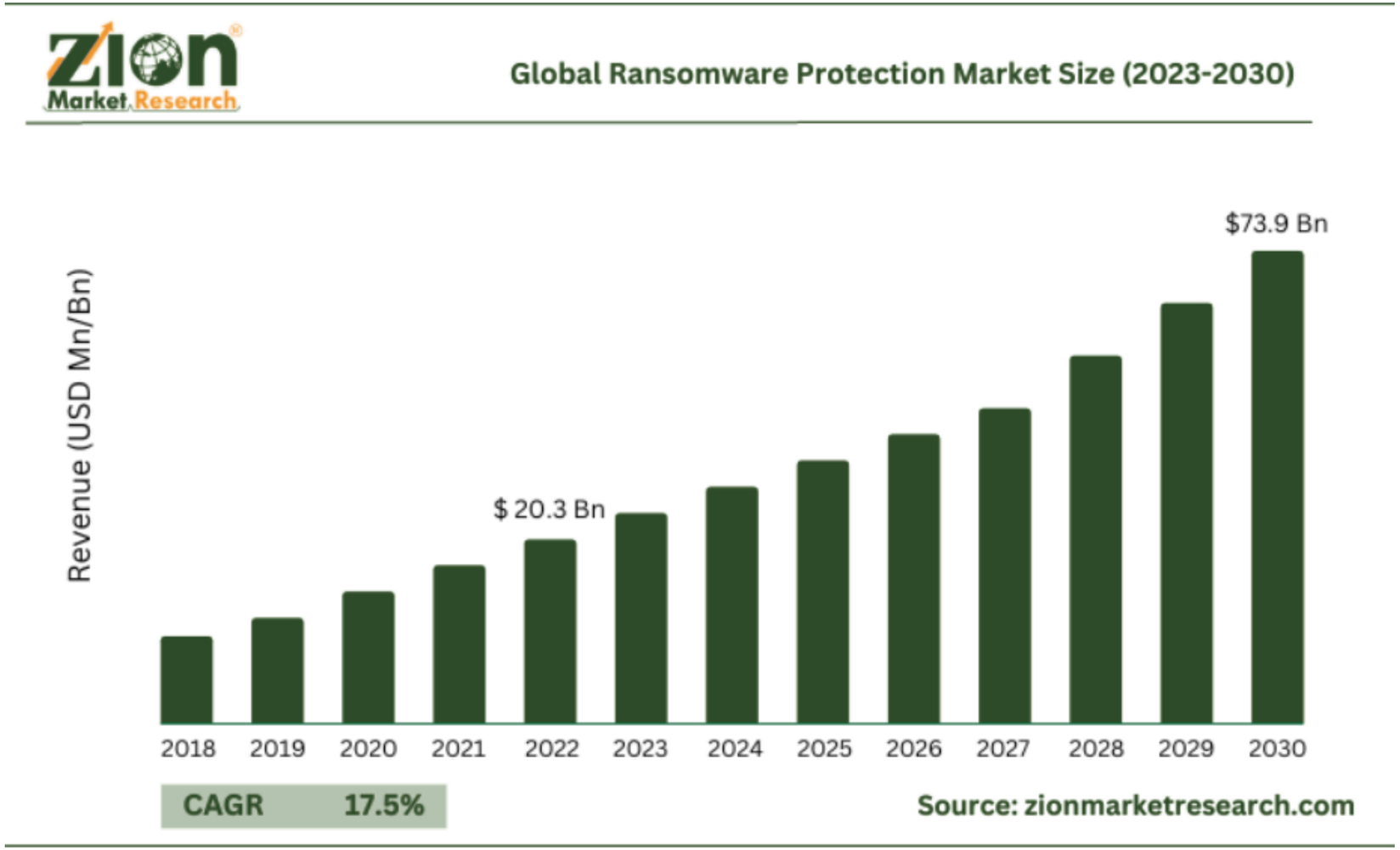
Presentation Agenda

- Ransomware Research Motivation
- Understanding ransomware concepts and encryption methods
- A new approach: interception during key generation
- Demonstration of PRNG hooking and decryption on a real ransomware sample
- Further Research and Next Steps
- References and Citations

vb 2025 BERLIN 24-26 Sept 2025

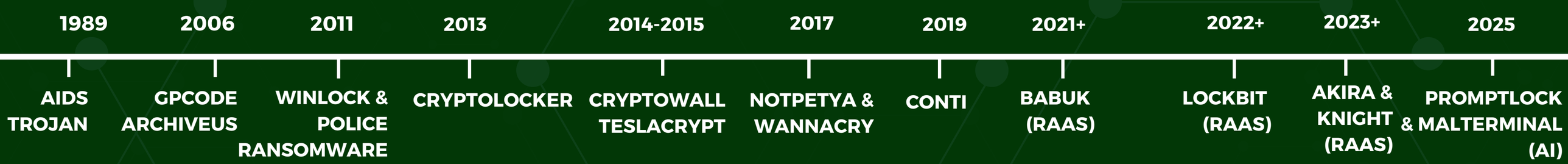
Motivation

- Ransomware Attacks Transformed a lot during these past years but crypto aspects stayed
- Double Exfiltration, Triple Exfiltration, AI Aspects...
- Victim is locked out until ransom is paid



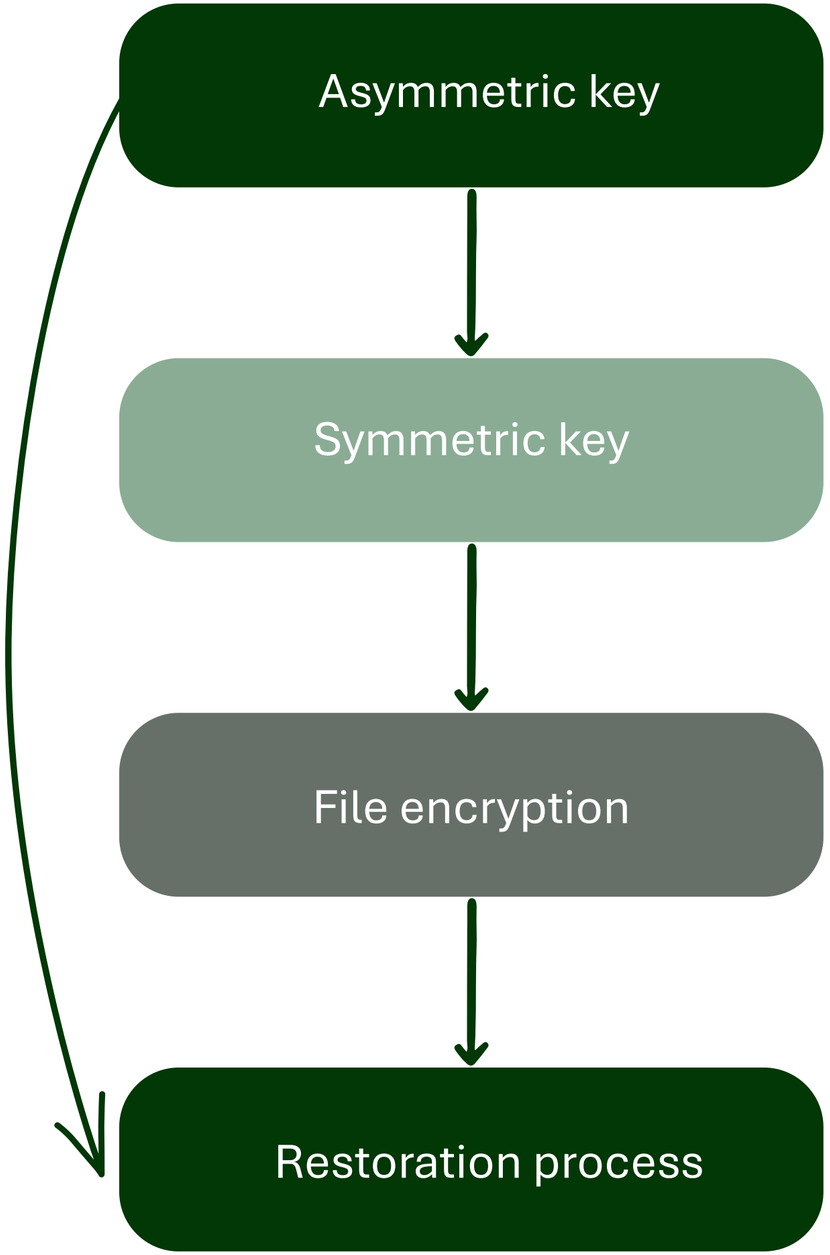
Motivation

Cryptographic locking is a consistent problem for over 36 years.

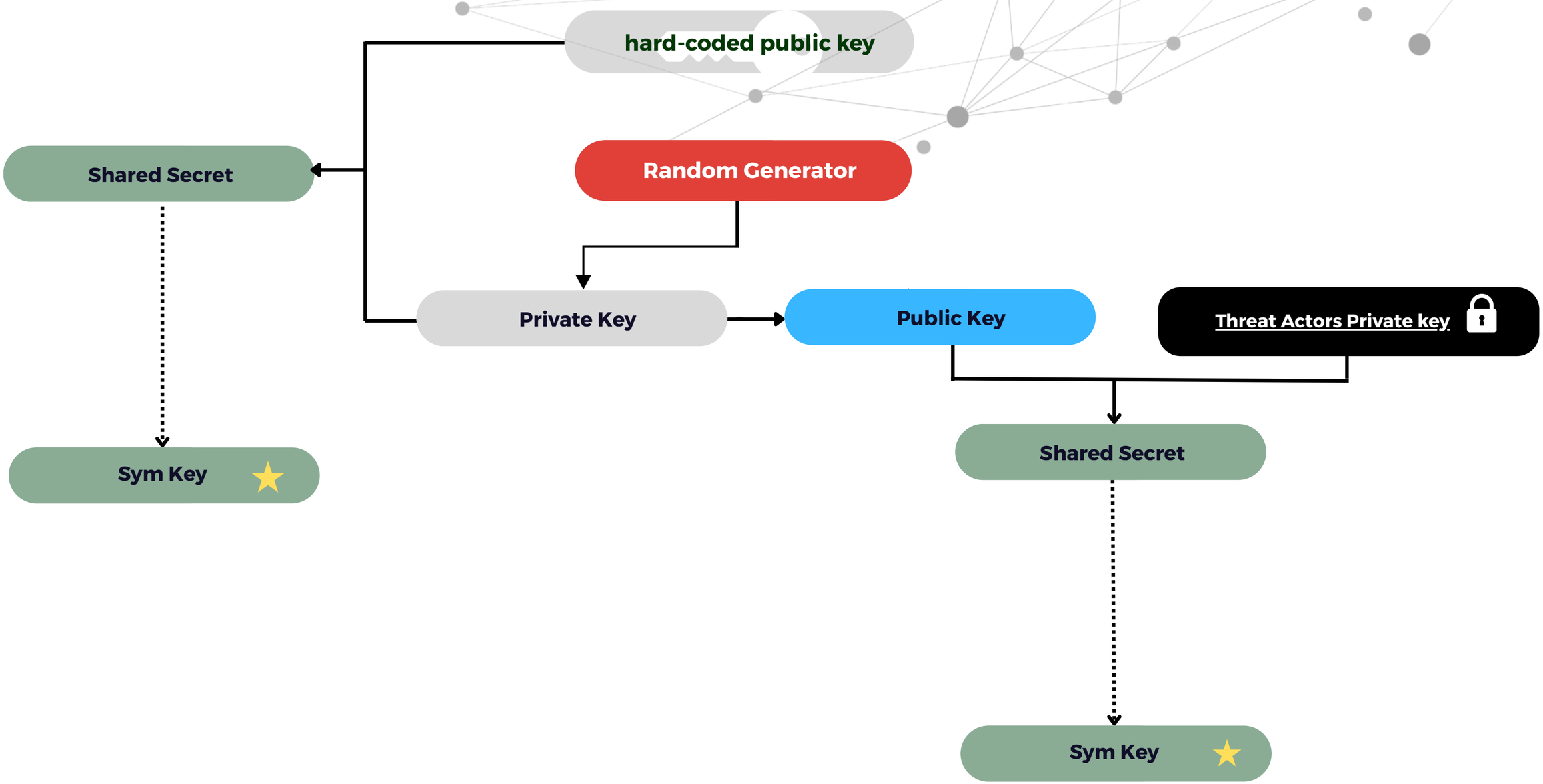


“State Of The Art” Ransomware Encryption Methods

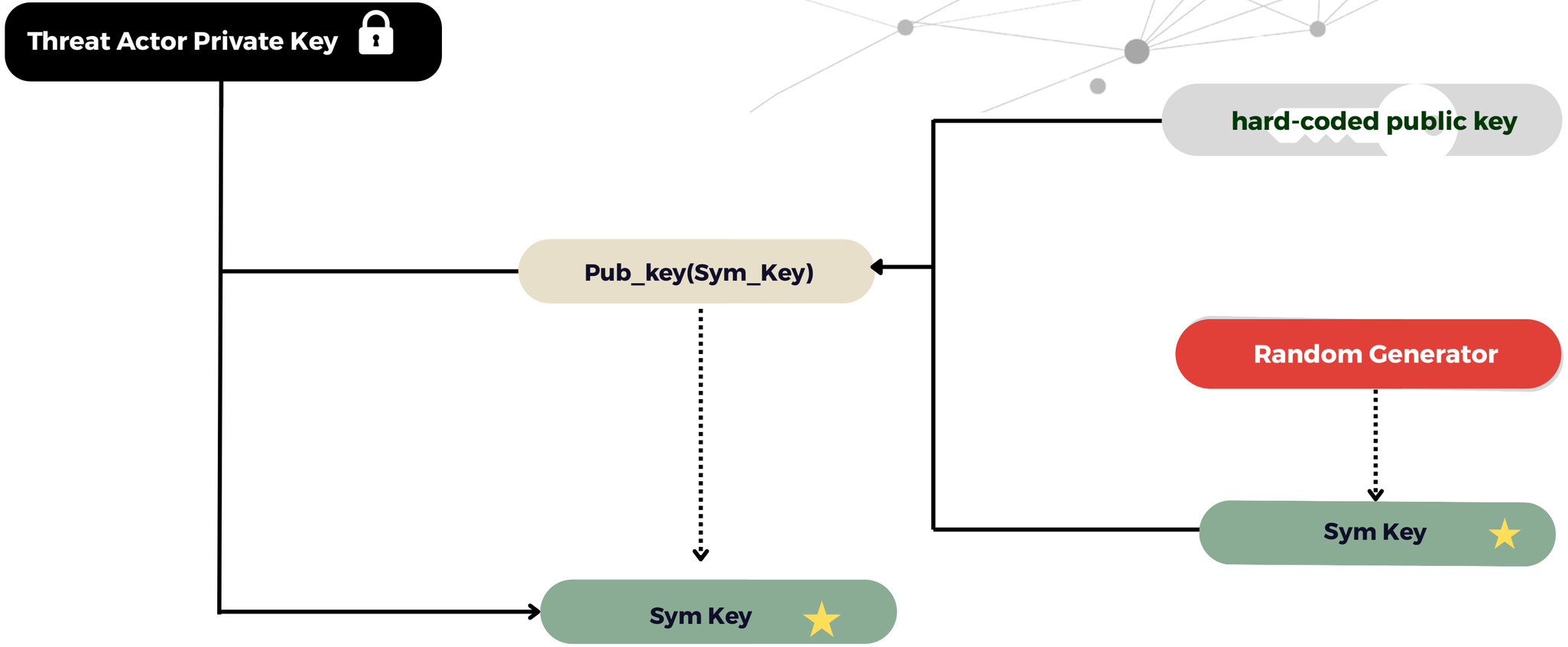
- Strong symmetric encryption with per-file unique keys
- PRNG generates seeds/keys
- Once key is discarded → practically unrecoverable without the threat actor’s private key
- We have to “trust” on finding a vulnerability



Encryption Mechanism 1



Encryption Mechanism 2



Rasomware Classification By Mechanism

Encryption Mechanism 1

Ransomhub
Knight
Babuk V1

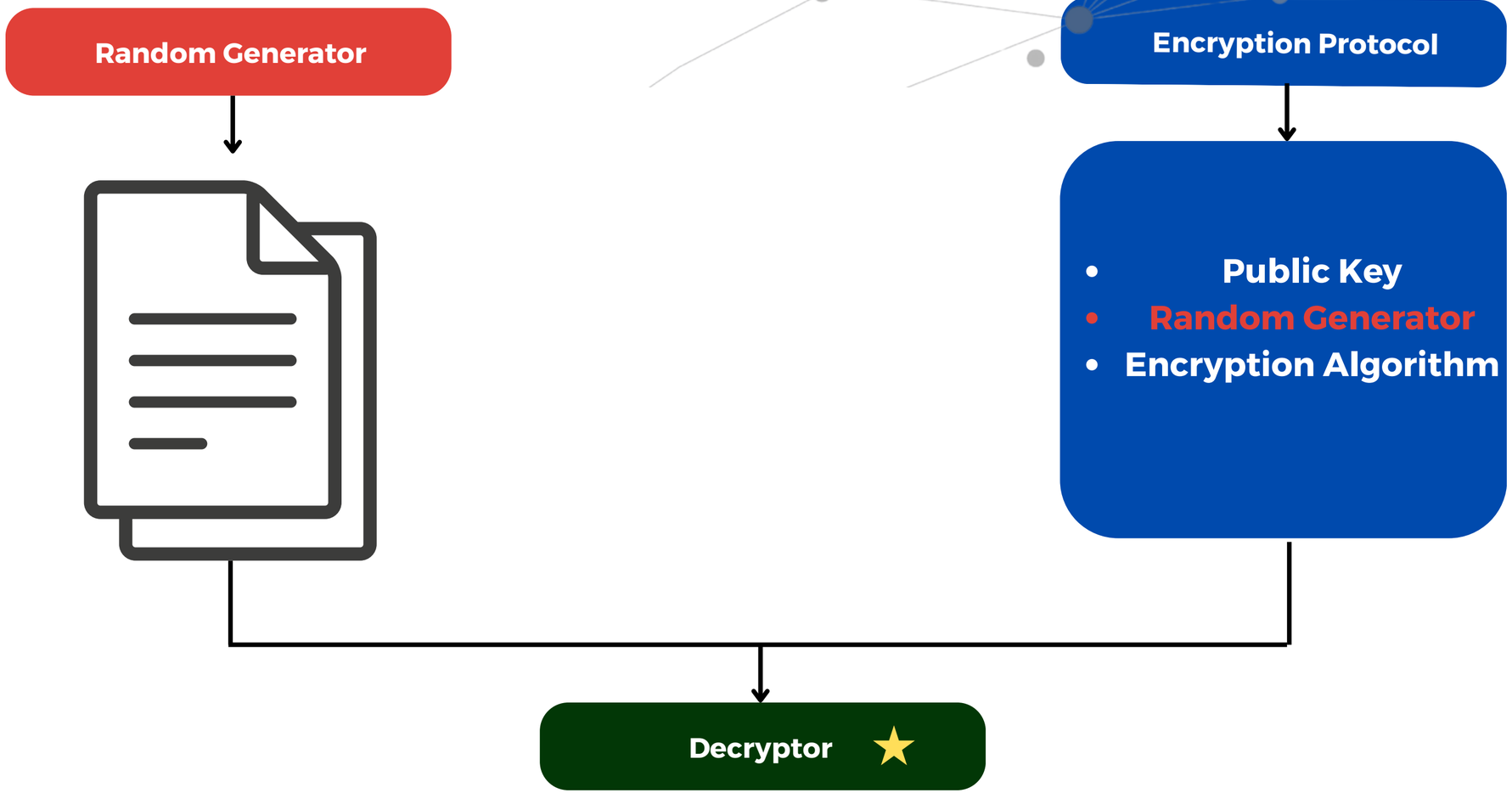
Encryption Mechanism 2

Akira
Darkside
Dragonforce
Rhysidia
Babuk V3
Homuwitch

Other

BianLian

Decryptor Generation



POC on real ransomware – Babuk V1:

- Babuk, also known as Babyk, is a Ransomware-as-a-Service (RaaS) family first observed in early 2021
- Following that year, the group’s builder and source code were leaked, which led to rapid multiple Babuk-derived variants targeting different operating systems
- The newer versions of Babuk (Babuk V3) drastically changed their encryption and distribution protocols
- Babuk V1 uses the First Encryption Methodology, V3 uses the Second.

Babuk V1 - Overview

- A non-obfuscated ransomware that utilizes common ransomware techniques such as multi-threading, standard encryption protocols and abusing Windows Restart Manager
- Babuk checks a list of hardcoded services and processes and kills them if they exist on the machine.
- Uses multi-threading per drive. For each thread, implements a classic folder iteration
- Generates a random value to agree on a shared secret and derives the symmetric encryption key from there

Babuk V1 – Encryption Protocol

1. The ransomware generates a PRNG (Pseudo Random Number Generator), using the RTLGenRandom function which is saved in the winAPI as SystemFunction036 in advapi32.dll.

```
dynamic_link_rtlgenrandom proc near

var_8= dword ptr -8
hModule= dword ptr -4

push    ebp
mov     ebp, esp
sub     esp, 8
push    offset CriticalSection ; lpCriticalSection
call    ds:InitializeCriticalSection
push    offset aAdvapi32Dll_0 ; "advapi32.dll"
call    ds:LoadLibraryA
mov     [ebp+hModule], eax
push    offset aSystemfunction ; "SystemFunction036"
mov     eax, [ebp+hModule]
push    eax             ; hModule
call    ds:GetProcAddress
mov     [ebp+var_8], eax
push    58h ; 'X'
push    offset chacha20key_1_from_Rand
call    [ebp+var_8]
mov     esp, ebp
pop     ebp
retn
dynamic_link_rtlgenrandom endp
```

Babuk V1 – Encryption Protocol

2. The 88 char random value is divided into - 32 bytes of a random key1, 12 bytes of a random nonce1, 32 bytes of a random key2 and 12 bytes of a random nonce2.

```
void __cdecl ECDH_Priv_keysetup(char *Priv_key, unsigned int priv_key_size)

unsigned int i; // [esp+0h] [ebp-4h]

EnterCriticalSection(&CriticalSection);
chacha8_xor((int)chacha20key_1_from_Rand, 20, (int)&chacha_nonce1, (int)&chacha_key2, (int)&chacha_key2, 44);
chacha8_xor(
  (int)&chacha_key2,
  20,
  (int)&chacha_nonce2,
  (int)chacha20key_1_from_Rand,
  (int)chacha20key_1_from_Rand,
  44);
for ( i = 0; i < priv_key_size; ++i )
  Priv_key[i] = chacha20key_1_from_Rand[i];
LeaveCriticalSection(&CriticalSection);
```

Babuk V1 – Encryption Protocol

3. Those are values to be used in a chacha algorithm:

- Chacha algorithm with key1 and nonce1 on key2_nonce2
- ChaCha algorithm with new_key2 and new_nonce2 on key1_nonce1
- Generating the 72 byte private key from the concatenation of newkey1_newnonce1_newkey2[:28]

As a result - a private key is derived.

```
void __cdecl ECDH_Priv_keysetup(char *Priv_key, unsigned int priv_key_size)

unsigned int i; // [esp+0h] [ebp-4h]

EnterCriticalSection(&CriticalSection);
chacha8_xor((int)chacha20key_1_from_Rand, 20, (int)&chacha_nonce1, (int)&chacha_key2, (int)&chacha_key2, 44);
chacha8_xor(
  (int)&chacha_key2,
  20,
  (int)&chacha_nonce2,
  (int)chacha20key_1_from_Rand,
  (int)chacha20key_1_from_Rand,
  44);
for ( i = 0; i < priv_key_size; ++i )
  Priv_key[i] = chacha20key_1_from_Rand[i];
LeaveCriticalSection(&CriticalSection);
```

Babuk V1 – Encryption Protocol

3. The private key derives a public key and a shared secret using the hard coded threat actor public key using ECDH.

Saves the public key generated in APPDATA in order to **be able to decrypt data for “paying clients”**

```
dynamic_link_rtlgenrandom();
ECDH_Priv_keysetup((int)&ECDH_priv_key, 72u);
ecdh_gen_keys(&ecdh_pub_key, &ECDH_priv_key);
ecdh_shared_sec(&ECDH_priv_key, HC_th_pub_key, &shared_secret);
sha256(&FINAL_KEY_1_FROM_SHARED, &shared_secret, 72);
sha256(&FINAL_KEY_FROM_2_SHARED, &shared_secret, 144);
memcpy(&final_nonce, &shared_secret, 12);
GetEnvironmentVariableW(L"APPDATA", Buffer, 0xF4u);
lstrcatW(Buffer, L"\\ecdh_pub_k.bin");
```

Babuk V1 – Encryption Protocol

4. From the shared secret, 2 Keys and a nonce are generated for the file encryption.

- **Key1 (32 bytes):** Result from sha256 on the first 72 bytes of the shared_secret
- **Key2 (32 bytes):** Result from sha256 on the first 144 bytes of the shared_secret
  Nonce (12 bytes): First 12 bytes of the shared_secret

```
dynamic_link_rtlgenrandom();
ECDH_Priv_keysetup((int)&ECDH_priv_key, 72u);
ecdh_gen_keys(&ecdh_pub_key, &ECDH_priv_key);
ecdh_shared_sec(&ECDH_priv_key, HC_th_pub_key, &shared_secret);
sha256(&FINAL_KEY_1_FROM_SHARED, &shared_secret, 72);
sha256(&FINAL_KEY_FROM_2_SHARED, &shared_secret, 144);
memcpy(&final_nonce, &shared_secret, 12);
GetEnvironmentVariableW(L"APPDATA", Buffer, 0xF4u);
lstrcatW(Buffer, L"\\ecdh_pub_k.bin");
```

Babuk V1 – Encryption Protocol

5. Per file encryption – Small files Algorithm (<±40MB):

```
{
  if ( FileSize.QuadPart <= 41943040 )        // Small file encryption (less than 40mb)
  {
    if ( FileSize.QuadPart > 0 )
    {
      v12 = MapViewOfFile(hFileMappingObject, 0xF001Fu, 0, 0, FileSize.LowPart);
      if ( v12 )
      {
        chacha8_xor((int)FINAL_KEY_1_FROM_SHARED, 20, (int)&final_nonce, (int)v12, (int)v12, FileSize.LowPart);
        chacha8_xor((int)FINAL_KEY_FROM_2_SHARED, 20, (int)&final_nonce, (int)v12, (int)v12, FileSize.LowPart);
        UnmapViewOfFile(v12);
      }
    }
  }
```

The file is encrypted with the same ChaCha described above twice:

- First key + Nonce on the file content
- Second key + Nonce on the encrypted result from the first ChaCha
  * 20 is the number of rounds and the counter for the ChaCha algorithm.

Babuk V1 – Encryption Protocol

5. Per file encryption – Big files Algorithm (>±40MB):

```
else                                    // Big file encryption
{
  v2 = sub_407200(FileSize.LowPart, FileSize.HighPart, 10485760, 0);// Divide by 10MB
  v5 = sub_407200(v2, HIDWORD(v2), 3, 0);   // Divide by 3
  for ( j = 0i64; j < 3; ++j )
  {
    v3 = sub_4072B0(j, HIDWORD(j), v5, HIDWORD(v5));
    dwFileOffsetLow = sub_4072B0(v3, HIDWORD(v3), 10485760, 0);
    lpBaseAddress = MapViewOfFile(
                      hFileMappingObject,
                      0xF001Fu,
                      HIDWORD(dwFileOffsetLow),
                      dwFileOffsetLow,
                      0xA00000u);
    if ( lpBaseAddress )
    {
      chacha8_xor(
        (int)FINAL_KEY_1_FROM_SHARED,
        20,
        (int)&final_nonce,
        (int)lpBaseAddress,
        (int)lpBaseAddress,
        10485760);
```

```
        chacha8_xor(
          (int)FINAL_KEY_FROM_2_SHARED,
          20,
          (int)&final_nonce,
          (int)lpBaseAddress,
          (int)lpBaseAddress,
          10485760);
        UnmapViewOfFile(lpBaseAddress);
      }
    }
  }
  CloseHandle(hFileMappingObject);
```

- File is Divided by 10MB and then by 3
- Only 3 chunks, equally distributed, consisting of 10MB each – are encrypted
- For each chunk, the encryption is identical to the small files encryption algorithm

In any case, at most ±40MB will be encrypted

Usermode API hooking - RTLGenRandom

Using the minhook opensource library – we could easily create a POC that hooks the function and saves the random values.

Syntax

C++ Copy

```
BOOLEAN RtlGenRandom(
  [out] PVOID RandomBuffer,
  [in]  ULONG RandomBufferLength
);
```

```
82  BOOLEAN WINAPI ProxySystemFunction036(PVOID RandomBuffer, ULONG RandomBufferLength)
83  {
84      int cx = 0;
85      cx = fwprintf_s(logger.GetFile(), L"%s\n", L"[HOOK] Intercepted call to SystemFunction036 (RtlGenRandom)");
86      cx = fwprintf_s(logger.GetFile(), L"%s: %lu\n", L"[HOOK] RandomBufferLength", (unsigned long)RandomBufferLengt
87      fflush(logger.GetFile());
88
89      BOOLEAN result = fpSystemFunction036(RandomBuffer, RandomBufferLength);
90
91      cx = fwprintf_s(logger.GetFile(), L"%s: %d\n", L"[HOOK] SystemFunction036 result", (int)result);
92      fflush(logger.GetFile());
93
94      return result;
95  }
```

Parsing only relevant results into a designated file:

1. Only 88 byte random generated values
2. Only the value itself
3. Bruteforcing if needed

POC

- Gets an encrypted file as an input.
- Gets the file with the saved random value generated from the hooker.
- Divide the generated value to: (key1, nonce1) and (key2,nonce2) tuples.

```
def main():
    if len(sys.argv) != 2:
        print("Usage: python pwn_babuk.py <encrypted_file>")
        sys.exit(1)

    encrypted_file = sys.argv[1]
    if not os.path.exists(encrypted_file):
        print(f"Error: Encrypted file '{encrypted_file}' not found!")
        sys.exit(1)

    # Read random data from rtkGenRandom Saved Data
    with open('rtlGenRandom_88.bin', 'rb') as f:
        random_data = f.read(144)

    data_88 = random_data[:88]
    key1 = data_88[:32]
    nonce1 = data_88[32:44]
    key2 = data_88[44:76]
    nonce2 = data_88[76:88]
```

POC

Creating the private key according to the algorithm described:

```
# First ChaCha20 transformation
key2_nonce2_concat = key2 + nonce2
encrypted_key2_nonce2 = chacha20_encrypt_decrypt(key2_nonce2_concat, key1, nonce1)
new_key2 = encrypted_key2_nonce2[:32]
new_nonce2 = encrypted_key2_nonce2[32:44]

# Second ChaCha20 transformation
key1_nonce1_concat = key1 + nonce1
encrypted_key1_nonce1 = chacha20_encrypt_decrypt(key1_nonce1_concat, new_key2, new_nonce2)
new_key1 = encrypted_key1_nonce1[:32]
new_nonce1 = encrypted_key1_nonce1[32:44]

# Create private key
private_key_bytes = new_key1 + new_nonce1 + new_key2[:28]

print(f"Private key: {private_key_bytes.hex()}")
```

POC

- Using the tiny-ECDH algorithm to generated the shared secret.

* The public key generation helps as a sanity check according to the public key saved in APPDATA.

```
77      # Load other party's public key
78      with open('hardcoded_pubkey_th.bin', 'rb') as f:
79          other_pub_key_data = f.read(144)
80
81      other_pub_words = np.frombuffer(other_pub_key_data[:(BITVEC_NWORDS*2*4)], dtype='<u4').copy()
82
83      # Compute shared secret
84      start_time = time.time()
85      shared_secret_raw = ecdh_shared_secret(local_private_key, other_pub_words)
86      shared_secret_time = time.time() - start_time
87      print(f"Shared secret computation: {shared_secret_time:.4f}s")
88      print(f"Shared secret: {shared_secret_raw.hex()}")
89      print(f"Shared secret length: {len(shared_secret_raw)} bytes")
90
91      # Extend shared secret
92      shared_secret_bytes = bytes(shared_secret_raw)
93      extended_data = shared_secret_bytes * 5
94      shared_secret = extended_data[:144]
```

POC

- Deriving the correct keys and the nonce for the decryption
- Using the ChaCha algorithms to decrypt the file

```
# Derive final keys
final_key1 = hashlib.sha256(shared_secret[:72]).digest()
final_key2 = hashlib.sha256(shared_secret).digest()
final_nonce = shared_secret[:12]

# Decrypt file
start_time = time.time()
with open(encrypted_file, 'rb') as f:
    encrypted_data = f.read()

first_decrypt = chacha20_encrypt_decrypt(encrypted_data, final_key2, final_nonce)
final_decrypt = chacha20_encrypt_decrypt(first_decrypt, final_key1, final_nonce)

decrypt_time = time.time() - start_time
print(f"Decryption time: {decrypt_time:.4f}s")
```

* The POC shown here doesn’t handle files bigger than ±40MB

Result

Running the code on a real encrypted file from a VX-underground sample which encrypted the “alias” file from the WSL folder.

Profit!

```
Step 1: Reading random data...
Extracted: key1(32), nonce1(12), key2(32), nonce2(12)
Step 2: First ChaCha20 transformation...
New key2: 32 bytes, New nonce2: 12 bytes
Step 3: Second ChaCha20 transformation...
==== Derived Results ====
new_key1:        9d3df55acf58004b24e7e5f636e3dfe8fcc5d32c8a32de4ad924ad9067c5e2e7
nonce1:      78cf28d1a83b81dd64a48c07
key2:        cd2c1690db9bc1bcbc67cfa7805fc19d17bcb5405a9c334a322d2e8228cca35a
nonce2:      9efc784a30844e343af75058
=======================
Step 4: Creating private key...
Private key length: 72 bytes
Private key bytes:      9d3df55acf58004b24e7e5f636e3dfe8fcc5d32c8a32de4ad924ad9067c5e2e778cf28d1a83b81dd64a48c07
cd2c1690db9bc1bcbc67cfa7805fc19d17bcb5405a9c334a322d2e82
[1526021533 1258313935 4142262052 3906986806  752076284 1256075914
 2427266265 3890398567 3509112696 3716234152  126657636 2417372365
 3166804955 2815387580 2646695808 1085651991 1244896346 2184064306]
[1526021533 1258313935 4142262052 3906986806  752076284 1256075914
 2427266265 3890398567 3509112696 3716234152  126657636 2417372365
 3166804955 2815387580 2646695808 1085651991 1244896346 2184064306]
BITVEC_NWORDS: 18
Generated public key: 98be0f4a45ac3e44460179a778e409eb46981ef81d87141048789be3ee4e1d33b3912eab4e8a317d9f27a0d833
5fce63f5e6fbee388b1ad6eee6d6a5d8b224fc104facc68bf21d00614b754f1c9688402c4f6dc3b1de0b3a20d373af67cb96bc34112599ca
032270d1b77336dec01b758418cef5cc1dc57124d909fb1b7db83ed5c70e0b1537433fe68c0aae7113f201
Public key length: 144 bytes
Step 6: Computing shared secret with tiny-ECDH...
pub key hc: 4e1d205dbe383349729b447b727ebcabc6957c97a40c4fe3ed7e7d72b05c45a50f2d22bded7f3343c8d156545e17179c7a9c
b75d3c3f5e02c422615cf4104e59022fbf712efefd0459b72b2b2b9b97dc59945025a7b34e1a48882f4b6fd0d30d4497d36b515641f3eacf
9963ed2ef1f4507b67a9bc6249336ce7691e4807dfe0efef70424026f20be3c3cba3a638ce05
Shared secret length: 144 bytes
shared secret: 69f4933e2e867106581f36870c00477f784de913d9dec6f772247df22cfb7335c0537368f611532d01cf07fe9173d68bc
1137709dc053a519450b1a1db6f1400780685f9e0e29f010f51072dfb038bb6cddd91994594d63403df4ddfea644f92883eb6e4fca6d3be0
1d4316ab8d37366f6aa4d16a2cd9be876e927c5117a4f41c3a006b1d793fb9305ef93297d1a3e04
Step 7: Deriving final keys...
Final key1: 32 bytes
Final key2: 32 bytes
Final nonce: 12 bytes
Step 8: Decrypting file...
Decryption complete! Result saved to decrypted_file.txt
Decrypted data preview: b'#!/bin/sh\nbuiltin alias "$@"\n'...
```

Notes, Disclaimers and Remarks:

- This whitepaper shows only a POC for one variant.
- This whitepaper's POC is using a crafted user-mode hooker.
- This solution cannot scale for a production solution as a user-mode hooker.
- Extra research might include a better and safer way to save the results from the hooked functions.
- Extra research might include futuristic additional cryptographic encryptions and key sharing alternatives to cover an hermetic solution.

Further Research & Next Steps

- Kernel-mode Hooking
- Safe Values Storing
- Hermetic Solution to all kinds of encryption methods
- Cover all ransomware variants and families.

Gotta Catch 'Em All

References and Citations

- https://github.com/kokke/tiny-ECDH-c
- https://www.chuongdong.com/reverse%20engineering/2021/01/03/BabukRansomware/
- github.com/hildaboo/babukransomware
- https://github.com/Hildaboo/BabukRansomwareSourceCode
- https://github.com/danielsousaoliveira/tiny-ECDH-python.git
- https://github.com/jtquisenberry/MinHook_Dll_Injection_Hooking
- https://learn.microsoft.com/en-us/windows/win32/api/ntsecapi/nf-ntsecapi-rtlgenrandom
- https://fabcoin.pro/struct_cha_cha8.html
- https://vx-underground.org/Samples/Families/Babuk
  - **IOC: 8203c2f00ecd3ae960cb3247a7d7bfb35e55c38939607c85dbdb5c92f0495fa9**

Questions?

Thank you!