2025 BERLIN

24 - 26 September, 2025 / Berlin, Germany

INTERCEPTING ENTROPY: HOOKING PRNG TO RECOVER RANSOMWARE ENCRYPTION KEYS

Raviv Rachmiel

Draastic, Israel

www.virusbulletin.com

ABSTRACT

We leverage ransomware's core properties to defend against them. Modern ransomware combines strong cryptography with pseudo-random number generation (PRNG). They generate a unique key to encrypt each file, releasing them back to the victim after the ransom is paid. Our core insight for undoing these attacks is that by obtaining the random numbers used to create these keys, also called seeds, we can decrypt the files ourselves. Our novel approach thus includes 'hooking' PRNG operations, storing the seeds in a secure store, and once an attack is detected, we use the stored seeds to conduct our symmetric key restoration, allowing us to decrypt ransomware-locked files on the fly without the attacker's help. In some cases, the same technique opens the way to decode configuration data and network traffic.

RANSOMWARE ENCRYPTION PROTOCOL OVERVIEW

For ransomware, and especially RaaS (ransomware-as-a-service), runtime performance and efficiency are important KPIs. As the slowest part in the chain of a ransomware attack process is asymmetric encryption, any improvement to it would constitute a significant win.

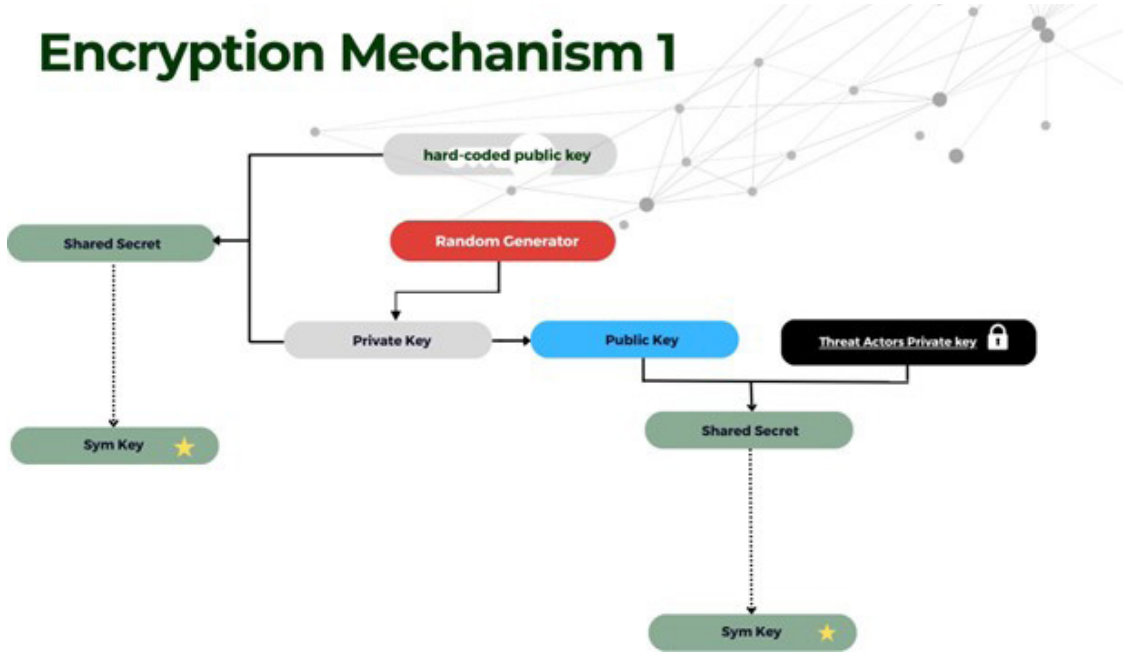
While ransomware has continued to evolve over the years, the core encryption methodologies have stayed the same. A common practice in ransomware development is equipping the fielded ransomware instance (installed on the victim) with some form of key shared with the operator. In turn, the instance can create a set of new per-attack or even per-encrypted-file symmetric encryption keys, encrypt them using the shared key, and send them to the operator.

Two main methods of designing an efficient encryption mechanism are in use by modern ransomware.

Encryption mechanism 1

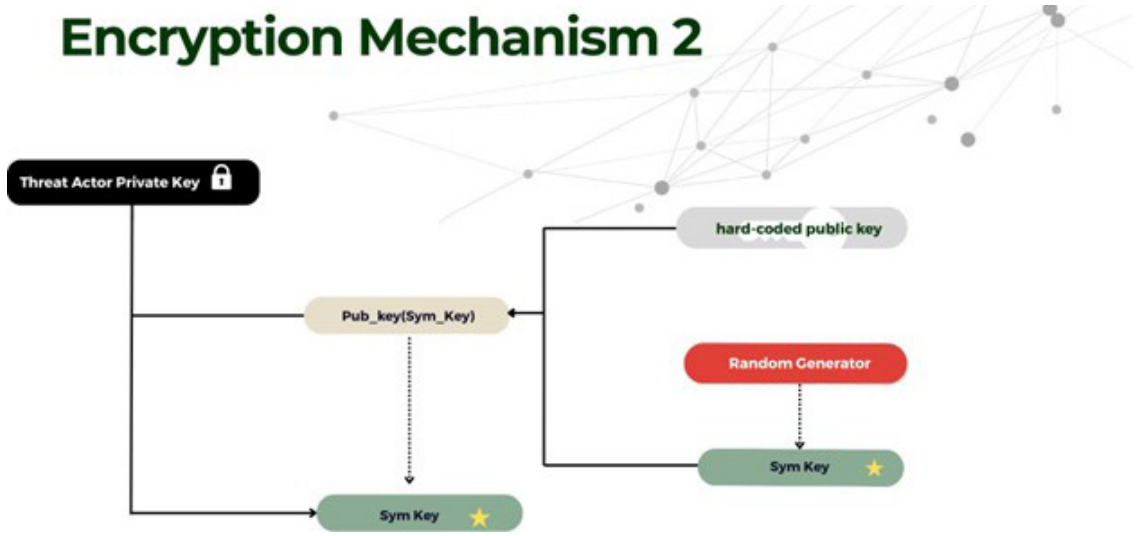
The first encryption mechanism involves using a hard-coded public key and generating a per-victim key pair by using an ECDH algorithm (or similar key agreement algorithms) to agree on a shared secret.

In this mechanism, a random value is generated on the victim's machine once, while the ransomware instance and the operator can generate a private-public key that is derived from that random value. When combining it with the operator's private-public keys, a shared secret can be agreed upon without explicitly sharing the private keys. In this case, the victim needs the public key of the ransomware operator, usually hard-coded in the executable binary, and the ransomware operator needs the public key generated by the victim, which is usually stored in a file or sent. After having a shared secret, the ransomware can derive symmetric keys for symmetric encryption protocols (AES, ChaCha, RC-4, etc.), and the threat actor can then derive the same keys for the decryption phase, using the shared secret.



Encryption mechanism 2

The second encryption mechanism involves generating a 'root' random value from which per-file symmetric keys are derived. Then, this key is encrypted using an asymmetric key and the result stored at the end of the encrypted file. These data slots at the end of encrypted files are sometimes called the encrypted file's metadata.



Examining both methodologies uncovers several overlapping behaviours:

1. Most of the encryption is performed using symmetric encryption.
2. The operator's public key employed throughout the process is embedded in the binary executable (and thus can be easily extracted from the sample).
3. A random value generation primitive is required.

We will leverage the third one to our advantage.

Modern OSs and frameworks offer several alternatives for random value generation, or more precisely, pseudo-random number generation.

Ransomware in the wild commonly uses one of the following PRNGs:

- cryptGenRandom (OS API)
- RtlGenRandom (OS API, a.k.a SystemFunction036) [1]
- Srand / rand (C functions)

Note that while the first two leave the task of generating random values to the OS, the last one accepts a seed and generates random values according to it. While srand is faster, the randomness of all generated values depends on the seed being random. Otherwise, restoring the seed allows for restoring the entire random value generation flow as they are derived from the seed. Thus, ransomware tools using srand must provide a random seed, and do so using a time, the result of a QueryPerformanceCounter call, or even chaining several different pseudo-random values.

Our core insight is that by collecting PRNG data used by the ransomware and reverse engineering the ransomware's encryption algorithm, we can find an approach to building a decryption tool per ransomware family. In this paper, we will examine the Babuk (V1 variant) RAAS tool that uses the RtlGenRandom function and encryption mechanism 1. The information and details we provide next are based on our independent reverse engineering of a dozen samples, and after developing a full proof-of-concept (PoC) to demonstrate our approach.

HOOKING PRNG FUNCTIONS OVERVIEW

Following our understanding of the value of obtaining random values used by ransomware, we conducted broad research on defence against ransomware.

Our theory was that using a user-mode hook to monitor the usage of PRNGs – RtlGenRandom, for example – we can save the generated pseudo-random data to later use it to decrypt encrypted files by ransomware, according to the encryption protocol examined by reverse engineering the ransomware variant.

BABUK V1 RANSOMWARE OVERVIEW

Babuk, also known as Babyk, is a RaaS family first observed in early 2021, known for targeting enterprises and other high-value ('big game') victims. Following that year, the group's builder and source code leaked, which led to the development of multiple Babuk-derived variants targeting different operating systems, including ESXi-focused lockers. That leak fundamentally changed the ransomware threat landscape as other actors reused and adapted Babuk's code. More importantly for us, many modern ransomware families have adopted the cryptographic methodologies and protocols originally implemented by Babuk.

Newer versions of Babuk (Babuk V3) drastically changed their encryption and distribution protocols, yet following their leakage, they also became widely used by other ransomware families.

Note that Babuk V1 uses encryption mechanism 1 (detailed above) while Babuk V3 uses encryption mechanism 2.

Technical overview

Babuk V1 is a non-obfuscated ransomware that utilizes common ransomware techniques such as multi-threading, standard encryption protocols, and abusing Windows Restart Manager to terminate processes that are using files in order to ensure successful encryption.

Babuk can work either with or without command-line parameters. Default operation (without arguments) leads to encrypting only the local machine, while providing arguments (namely 'lanfirst', 'lansecond' and 'nolan') controls the encryption of network drive files.

In terms of persistence and evasion, Babuk checks a list of hard-coded services, and if they are open, it stops and terminates them.

The ransomware does the same for processes, using the classic methods of CreateToolHelp32snapshot, Process32firstw, process32nextw, and TerminateProcess.

```
BOOL kill_procs()
{
  BOOL i; // [esp+0h] [ebp-240h]
  HANDLE hSnapshot; // [esp+4h] [ebp-23Ch]
  HANDLE hProcess; // [esp+8h] [ebp-238h]
  unsigned int j; // [esp+Ch] [ebp-234h]
  PROCESSENTRY32W pe; // [esp+10h] [ebp-230h] BYREF

  hSnapshot = CreateToolhelp32Snapshot(0xFu, 0);
  pe.dwSize = 556;
  for ( i = Process32FirstW(hSnapshot, &pe); i; i = Process32NextW(hSnapshot, &pe) )
  {
    for ( j = 0; j < 0x1F; ++j )
    {
      if ( !lstrcmpW((&lpString1)[j], pe.szExeFile) )
      {
        hProcess = OpenProcess(1u, 0, pe.th32ProcessID);
        if ( hProcess )
        {
          TerminateProcess(hProcess, 9u);
          CloseHandle(hProcess);
        }
        break;
      }
    }
  }
  return CloseHandle(hSnapshot);
}
```

Folder iteration

Folder iteration is also done using the same iteration protocol, and for each file, it uses the encryption protocol, which is further elaborated later in this paper.

Multi-threading

The ransomware chooses the number of threads to initiate based on the number of cores in the machine (doubles the number). Each thread applies the folder iteration and encryption algorithm to one drive (local or remote).

Encryption protocol

Babuk V1's encryption process uses a combination of Elliptic-Curve Diffie-Hellman (ECDH) for key exchange and the ChaCha8 stream cipher for file encryption.

First, as part of the process initialization, the ransomware generates the shared secret. These steps are performed using the RtlGenRandom function, mentioned above.

```
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
```

Using this generated random value, which includes 88 randomly generated characters, the ransomware generates a private key using the following algorithm. The 88-character random value is divided into:

- 32 bytes of a random key1
- 12 bytes of a random nonce1
- 32 bytes of a random key2
- 12 bytes of a random nonce2.

Those are values to be used in a ChaCha algorithm [2]:

```
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);
```

As we can see, we get the key1_rand (32), the nonce1 (12), key2_rand (32), nonce2 (12).

And we use the following algorithm:

- We use the ChaCha algorithm with key1 and nonce1 on the concatenation of key2_rand and nonce2 to get new key2_enc and nonce2_enc.
- Then we use ChaCha again, with the new key2_enc and nonce2_enc on a concatenation of key1_rand and nonce1 to generate new key1_enc and new_nonce1.

Out of the 88 bytes that changed, Newkey1enc_newnonce1_newkey2[:28] is the private_key for the ECDH – meaning a total of 72 bytes, which are the concatenation of the new encrypted key1, the new encrypted nonce1, and 28 bytes out of the newly encrypted key2 (32 bytes in total).

The constant 20 is the counter and number of rounds for the ChaCha algorithm.

Once Babuk has the private key, it generates the public key using the ECDH algorithm [3].

After the public key is generated, the ransomware generates a shared secret using the private key of the victim machine (generated by the PRNG) and the hard-coded public key of the ransomware operator. The private key is 72 bytes and the public key is 144 bytes.

The ECDH generation assures all numbers are correct and that the shared secret will be equal when generating it with Pub_Key_A and Priv_Key_B.

Once it has a shared secret, the ransomware computes a custom implemented SHA256 twice to generate two ChaCha8 keys for the file encryption.

The first SHA256 is computed on the first 72 bytes of the shared_secret, which is of size 144 bytes. The second SHA256 is computed on all of the 144 bytes.

After that, it generates a 12-byte nonce which is the first 12 bytes of the shared secret.

The ransomware stores the public key it generated on the victim machine in the 'ecdh_pub_k.bin' file, placed under the 'appdata' folder. This part is necessary because the ransomware operator has their own private key but they need the public key of the victim in order to generate the shared secret and to be able to later decrypt the files by generating the same keys and nonce.

```
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");
```

To sum up, the final resulting random values will be:

- 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

The per-file encryption process is performed as follows using two variants, one for 'small' files (<± 40MB) and another for larger files.

Small file encryption (less than 40MB)

- The file is encrypted with the same ChaCha8 as described above, twice. Once with the first key, generated by a SHA256 operation on the first 72 bytes of the shared secret, then with the second key, generated by an SHA256 operation on all of the 144 bytes of the shared secret. Both encryptions get the same nonce as an input, which is the first 12 bytes of the shared secret.
- As described before, the constant parameter 20 is the number of rounds and counter for the ChaCha8 encryption

```
{
  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);
      }
    }
  }
```

Larger file encryption: (greater than 40MB)

- For larger files, there is a preliminary check to calculate the file size divided by 10MB and then by 3. For large files, the ransomware encrypts only three parts of the file, equally distributed within the file, encrypting 10MB of each of the three parts.
- In total, there are 30MB of encrypted bytes.
- Each chunk is encrypted in the process detailed above for (whole) small files.

```
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);
```


```
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);
```


This file encryption algorithm ensures that, for each file, at most ±40MB will be encrypted, so the encryption doesn't take too long.

BABUK V1 RECOVERY POC

After we reverse-engineered Babuk's encryption protocol, we created a dedicated program for hooking RtlGenRandom, aiming to capture the 88-byte random seeds to a file. Next, we detail how we used the captured seeds log to develop a decryption tool.

The decryption tool

The decryption tool takes an encrypted file as input and reads the saved 88-byte random value from the hook log file. It splits this 88-byte value into the four components (key1_rand, nonce1, key2_rand, nonce2) exactly as the ransomware did.

```python
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]
```

Next, it creates the private key according to the explanation above regarding the encryption: it computes the 72-byte ECDH private key using the same ChaCha8-based routine as described earlier.

```python
# 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()}")
```

After the private key generation, the decryptor uses the tiny-ECDH algorithm to generate the shared secret (and the public key just for a sanity check, according to the public key saved in APPDATA, but this part is not necessary for a correct flow).

```python
    # Load other party's public key
    with open('hardcoded_pubkey_th.bin', 'rb') as f:
        other_pub_key_data = f.read(144)

    other_pub_words = np.frombuffer(other_pub_key_data[:(BITVEC_NWORDS*2*4)], dtype='<u4').copy()

    # Compute shared secret
    start_time = time.time()
    shared_secret_raw = ecdh_shared_secret(local_private_key, other_pub_words)
    shared_secret_time = time.time() - start_time
    print(f"Shared secret computation: {shared_secret_time:.4f}s")
    print(f"Shared secret: {shared_secret_raw.hex()}")
    print(f"Shared secret length: {len(shared_secret_raw)} bytes")

    # Extend shared secret
    shared_secret_bytes = bytes(shared_secret_raw)
    extended_data = shared_secret_bytes * 5
    shared_secret = extended_data[:144]
```

Using the shared secret, we extract the keys and the nonce according to the algorithm. This is also what the ransomware operator would do when they want to decrypt the files for ‘paying customers’.

After we have all the data, we will decrypt the file with a symmetric approach according to the encryption, because ChaCha is a symmetric encryption algorithm.

```
# 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 here doesn't handle files bigger than ±40MB, but it is very clear how to implement it according to the explanation described in the previous sections.

The result would be written to a decrypted file, and a preview will be printed to the console.

For example, for an encrypted alias file, from the WSL folder (encrypted on a virtual machine by the original sample from VX-underground [4]):

```
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'...
```

We can see that the result is a readable text that suits the alias file from the WSL folder.

MODIFIED MINHOOK POC EXPLANATION

For the POC, we decided to use the open-source MinHook project, designed to provide a POC for *Windows* API hooking and DLL injection as a 'minimalistic API hooking library for *Windows*'. That way, we could easily create a POC that hooks the RtlGenRandom function, saves the RandomBufferLength and the result in a different file, and returns the correct value.

The following is a screenshot of the function syntax according to the official *Windows* API documentation:

Syntax

C++ Copy

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

The hooked function is written as follows:

```
BOOLEAN WINAPI ProxySystemFunction036(PVOID RandomBuffer, ULONG RandomBufferLength)
{
    int cx = 0;
    cx = fwprintf_s(logger.GetFile(), L"%s\n", L"[HOOK] Intercepted call to SystemFunction036 (RtlGenRandom)");
    cx = fwprintf_s(logger.GetFile(), L"%s: %lu\n", L"[HOOK] RandomBufferLength", (unsigned long)RandomBufferLength
    fflush(logger.GetFile());

    BOOLEAN result = fpSystemFunction036(RandomBuffer, RandomBufferLength);

    cx = fwprintf_s(logger.GetFile(), L"%s: %d\n", L"[HOOK] SystemFunction036 result", (int)result);
    fflush(logger.GetFile());

    return result;
}
```

This function runs instead of the RtlGenRandom function (SystemFunction036) but returns the same result.

The hook writes the results to a log file with entries such as:

```
[HOOK] Intercepted call to SystemFunction036 (RtlGenRandom)
[HOOK] RandomBufferLength: RANDOM_VALUE_LENGTH
[HOOK] SystemFunction036 result: RANDOM_VALUE
```

```
C:\Users\Rick\source\repos\MinHook_Dll_Injection_Hooking\x64\Debug>Injector.exe --pid 3984 --dll Payload1.dll
ARGUMENTS
Argument: --pid
Argument: 3984
Argument: --dll
Argument: Payload1.dll

*** Dll was successfully injected. ***

*** End of injector application. ***
```

From this file, we generate files per result that hold only the random values that have a length of 88 bytes.

There should be only one file generated, because there should be only one result with a size of 88 bytes. If there were more files, we would have to brute-force them to find the right encryption keys.

ADDITIONAL RANSOMWARE RESEARCH

To further understand the potential of this solution, we checked the PRNG mechanisms of some more ransomware tools. We tried classifying the ransomware according to whether it has encryption mechanism 1 (described above), encryption mechanism 2, or neither.

The following is a list of all ransomware tools investigated:

1. Akira – uses encryption keys generated by QueryPerformanceCounter and uses encryption mechanism 2.
2. Ransomhub – uses ECDH and derives a symmetric key from it; uses encryption mechanism 1.
3. Knight – just like Ransomhub, uses ECDH (curve25519). Generates a random number based on crypto/rand; uses encryption mechanism 1.
4. BianLian – uses GO encryption with a standard AES-CBC. Key and IV are hard coded. Uses none of the encryption mechanisms, but does use a function that can be hooked before.
5. Darkside – uses AutoSeededRandomPool::Reseed (derived from /dev/urandom); uses encryption mechanism 2.
6. Dragonforce – uses encryption mechanism 2 using CryptGenRandom.
7. Homuwitch – generated using PBKDF2. Uses encryption mechanism 2.
8. Rhysidia – uses a ChaCha20 key generation and then saves it via a hard-coded public RSA key for each sample. Uses encryption mechanism 2.

9. Babuk V1 – as described in the article, uses encryption mechanism 1.
10. Babuk V3 – invokes cryptGenRandom for each file; uses encryption mechanism 2.
11. Medusa – uses srand with a pseudo-random seed and then saves the key in the metadata, encrypted with the RSA public key. Uses encryption mechanism 2.

In summary, of these 11 ransomware families, seven use encryption mechanism 2, three use mechanism 1, and one uses neither. In other words, the vast majority rely on PRNG-derived keys or secrets, which suggests that a hooking strategy could potentially be effective against many of them.

NOTES, DISCLAIMERS, AND REMARKS

- This paper demonstrates a proof-of-concept on one specific ransomware variant (Babuk V1).
- Our PoC uses a custom user-mode hook. While effective for demonstration, a user-mode hook may not be a practical or comprehensive defence solution in real-world deployments.
- This solution cannot scale for a production solution as a user-mode hooker.
- Additional research is necessary to ensure that captured data (seeds/keys) is stored safely and cannot be tampered with or leaked during an attack.
- Additional research might explore more comprehensive solutions that cover a broader range of cryptographic methods and key-sharing schemes, to make the defensive approach as accurate as possible against different ransomware techniques.

FURTHER RESEARCH AND NEXT STEPS

- Investigate kernel-mode hooking to intercept PRNG calls at a lower level.
- Develop secure logging/storage mechanisms for captured seeds and keys (to prevent the attacker from detecting or altering the saved data).
- Extend the approach to handle varied cryptographic schemes, so the solution remains effective as ransomware evolves. Evaluate the technique across all known ransomware families to ensure it works universally and identify any that use PRNG in unexpected ways.

ACKNOWLEDGEMENTS

Last but not least, I want to take the opportunity to thank some contributors without whom my paper wouldn't have reached the goals and achievements it deserves, from explainability to the full source and inspiration of my VB2025 presentation:

- Dr Yaniv David, Technion, IL
- Aviad Shamriz, IL

Thank you for always being a part of my journey and specifically for helping me with this research and paper.

REFERENCES & BIBLIOGRAPHY

[1] Microsoft. RtlGenRandom function (ntsecapi.h). Windows App Development. https://learn.microsoft.com/en-us/windows/win32/api/ntsecapi/nf-ntsecapi-rtlgenrandom.

[2] ChaCha8 Class Reference. Fabcoin Core. https://fabcoin.pro/struct_cha_cha8.html.

[3] kokke / tiny-ECDH-c. https://github.com/kokke/tiny-ECDH-c.

[4] Babuk. VX-underground. https://vx-underground.org/Samples/Families/Babuk.

[5] Chuong Dong. Babuk Ransomware. https://www.chuongdong.com/reverse%20engineering/2021/01/03/BabukRansomware/.

[6] Hildaboo / BabukRansomware. https://github.com/hildaboo/babukransomware.

[7] Hildaboo / BabukRansomwareSourceCode. https://github.com/Hildaboo/BabukRansomwareSourceCode.

[8] danielsousaoliveira / tiny-ECDH-python. https://github.com/danielsousaoliveira/tiny-ECDH-python.git.

[9] jtquisenberry / MinHook_Dll_Injection_Hooking. https://github.com/jtquisenberry/MinHook_Dll_Injection_Hooking.

IOC

8203c2f00ecd3ae960cb3247a7d7bfb35e55c38939607c85dbdb5c92f0495fa9