CRACKING THE CHAINS

Accelerating Ransomware Recovery via LLM-Assisted Engineering and Verification

SungWook Jang · YoungMook Kang
Financial Security Institute (FSI) — CSIRT

WHO WE ARE

SungWook Jang

Malware Analysis · Incident Response

Financial Security Institute — CSIRT

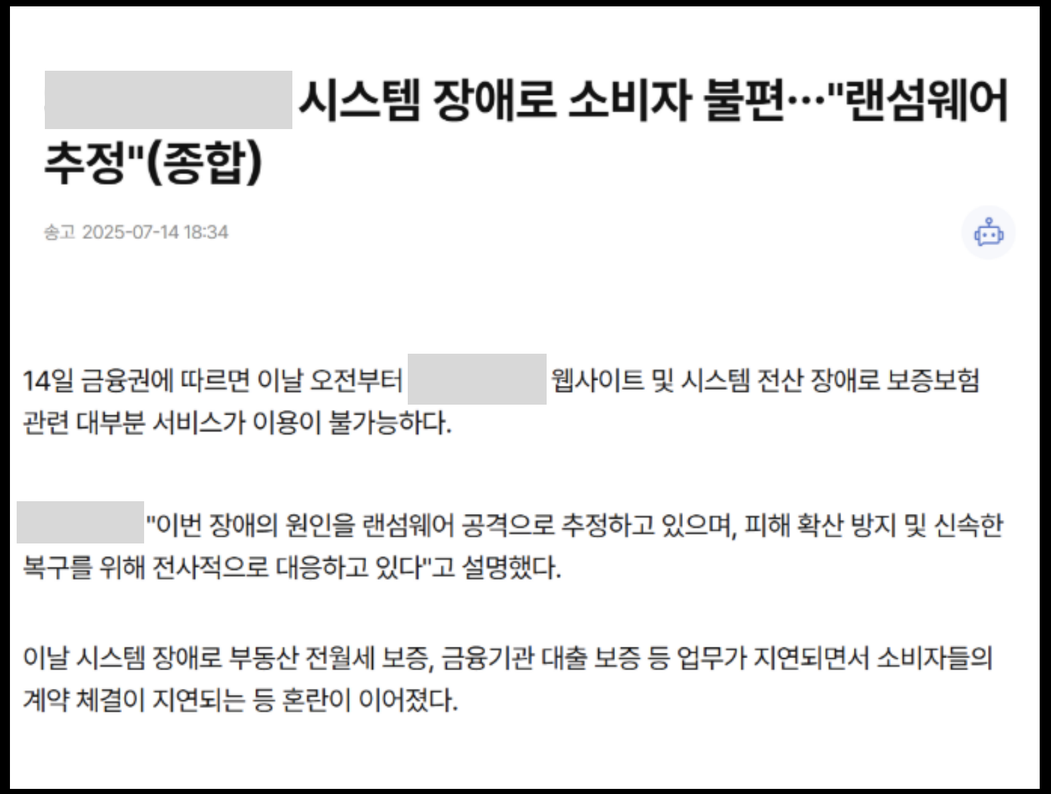
YoungMook Kang

Malware Analysis · Incident Response

Financial Security Institute — CSIRT

A GUARANTEE SERVICE GOES DARK

Before any analysis, the public already felt it — the incident hit the news the same day:



[redacted] 시스템 장애로 소비자 불편…"랜섬웨어 추정"(종합)

송고 2025-07-14 18:34

14일 금융권에 따르면 이날 오전부터 [redacted] 웹사이트 및 시스템 전산 장애로 보증보험 관련 대부분 서비스가 이용이 불가능하다.

[redacted] "이번 장애의 원인을 랜섬웨어 공격으로 추정하고 있으며, 피해 확산 방지 및 신속한 복구를 위해 전사적으로 대응하고 있다"고 설명했다.

이날 시스템 장애로 부동산 전월세 보증, 금융기관 대출 보증 등 업무가 지연되면서 소비자들의 계약 체결이 지연되는 등 혼란이 이어졌다.

PRESS · JULY 14, 2025

Major guarantee insurer's outage disrupts customers — "suspected ransomware attack"

"We are treating this as a suspected ransomware attack and mounting a company-wide response to ensure a swift recovery."

— The affected insurer, 14 July 2025

Behind the outage: a hands-on ransomware attack — **and the response that brought it all back**

HOW WE GOT THE DATA BACK

A tight loop of problem and solution — **analyze → find the flaw → build the tool → recover**

DAY 1
INITIAL RESPONSE

Malware teardown and attribution
— under the pressure of a live national outage

D1

MALWARE TEARDOWN - WHAT enc DOES

CORE CAPABILITIES

- Generates the ChaCha20 key & nonce
- --limit — max bytes to encrypt
- --ratio — partial-encryption interval
- --device — raw disk vs file targets
- Skips R3ADM3.txt note & .ENCRT files

```
if ( argc == 5 )
{
  v25 = 0LL;
  v24 = 0LL;
  v23 = -1;
  v22 = -1;
  for ( i = 1; i < 5; ++i )
  {
    if ( !(unsigned int)strncmp(argv[i], "--device=", 9LL) )
    {
      envp = (const char **)(8LL * i);
      v25 = (char *)(*(const char **)((char *)argv + (_QWORD)envp) + 9);
    }
    else if ( !(unsigned int)strncmp(argv[i], "--keystore=", 11LL) )
    {
      envp = (const char **)(8LL * i);
      v24 = *(const char **)((char *)argv + (_QWORD)envp) + 11;
    }
    else if ( !(unsigned int)strncmp(argv[i], "--ratio=", 8LL) )
    {
      v23 = atoi(argv[i] + 8);
    }
    else
    {
      if ( (unsigned int)strncmp(argv[i], "--limit=", 8LL) )
        goto LABEL_2;
      v22 = atoi(argv[i] + 8);
```

Decompiled option parser — argc, --device/--keystore/--ratio/--limit

WHAT enc TARGETS — AND SKIPS

The --exts option drives three code branches — plus two hard-coded skip rules:

THREE TARGETING MODES

- all — every file, no filter
- disk — raw block device, low-level
- <ext-list> — up to 32 extensions

TWO SKIP RULES

- R3ADM3.txt — its own ransom note
- *.ENCRT — already-encrypted files
- (prevents double-encryption)

```
if ( !strcasecmp(name, "R3ADM3.txt") ) { puts("Skipping R3ADM3.txt file"); return; }
```

ATTRIBUTION: THE GUNRA GROUP

The ransom note, the .ENCRT extension and TTPs point to **Gunra**

```
# ################################################################################
# !!! The server has been encrypted by ransomware. !!!
# What happened?
# First, We completely stopped the cluster and data services using the shell below and stopped I/O to the
data disk.
#
# srvctl stop database -d <instance_name>
# srvctl stop asm
# crsctl stop cluster -all
#
# Then, encrypted the data disk using a low-level disk encryption ransomware tool. The algorithm used at this
time was RSA 4096, which is virtually impossible to decrypt without the secret key.
# We only encrypted the operational database servers among your servers. Other app servers are not affected.
#
# !!! Caution !!!
# You can try to restore the service yourself, but be sure to pay close attention to the following:
# Do not perform any arbitrary operations that can initiate I/O to the data disk. The data disk is currently
encrypted and cannot be used for normal service. In this state, actions such as starting the service can
destroy the integrity of the encrypted data on the data disk and make decryption impossible.
#
# The tools used for ransomware and various logs can be checked in the /root/crypt directory. There are RSA
public keys and encryption progress patterns stored there. Do not delete or modify the contents of the
/root/crypt directory. We need the contents stored in that directory for decryption.
#
# How can I recover it?
# You can get RSA private keys and a dedicated tool for ransom recovery from us, and we can prove that we can
decrypt it upon request. Please contact the address below. We demand money and accept cryptocurrencies BTC
and Monero. Your CEO or CFO can contact with us.
```

IDENTIFIED FROM

- Ransom note left as issue.net
- R3ADM3.txt marker filename
- Encrypted extension .ENCRT
- RSA-4096 + ChaCha20 scheme
- Hands-on artifacts in /root/crypt

STAGING GROUND: /root/crypt

One directory dropped on every server — **no C2, everything needed for the attack.**

Name	Size	Compre..	Modified Date
enc	124 232	50 416	2025-07-14 03:48
issue.net	1 903		2025-07-14 03:48
nohup.out	236		2025-07-14 04:06
public.pem	800		2025-07-14 03:48
s	239		2025-07-13 00:06
_dev_sdc1.keystore	512		2025-07-14 03:50
_dev_sdc1.progress	8		2025-07-14 04:06

enc (encryptor) · issue.net (note) · public.pem (RSA-4096) · s (script) · .keystore · .progress

PLAN VS REALITY: THE SCRIPT 's'

The staged script tells one story — the binary that actually ran tells another

THE PLANNED COMMAND (script 's')

```
mv -f /etc/issue.net /root/crypt/issue.net.bak
mv -f /root/crypt/issue.net /etc/
srvctl stop database -d PIFM
srvctl stop asm
crsctl stop cluster -all
nohup /root/crypt/enc --device=/dev/sdc1 --keystore=/root/crypt --ratio=2 --limit=100 &
```

BUT THE BINARY REJECTED THOSE OPTIONS

```
root@localhost:~/crypt# pwd
/root/crypt
root@localhost:~/crypt# ./enc --device=/dev/sdc1 --keystore=/root/crypt --ratio=2 --limit=100
error: unknown option `--device=/dev/sdc1`
Usage: encryptor --threads=<threads> --path=<input> --exts=<extensions> --ratio=<ratio> --keyfile=<keyfile>
```

`unknown option --device` — the executed binary was a different build than the one they prepared

DAY 2
THE FATAL FLAW

Two lines of broken C turned an unbreakable scheme into a 256-try guess

D2

HYBRID CRYPTO — BUILD FAST

The ChaCha20 keystream XORs the disk; the key is sealed with RSA — normally unrecoverable

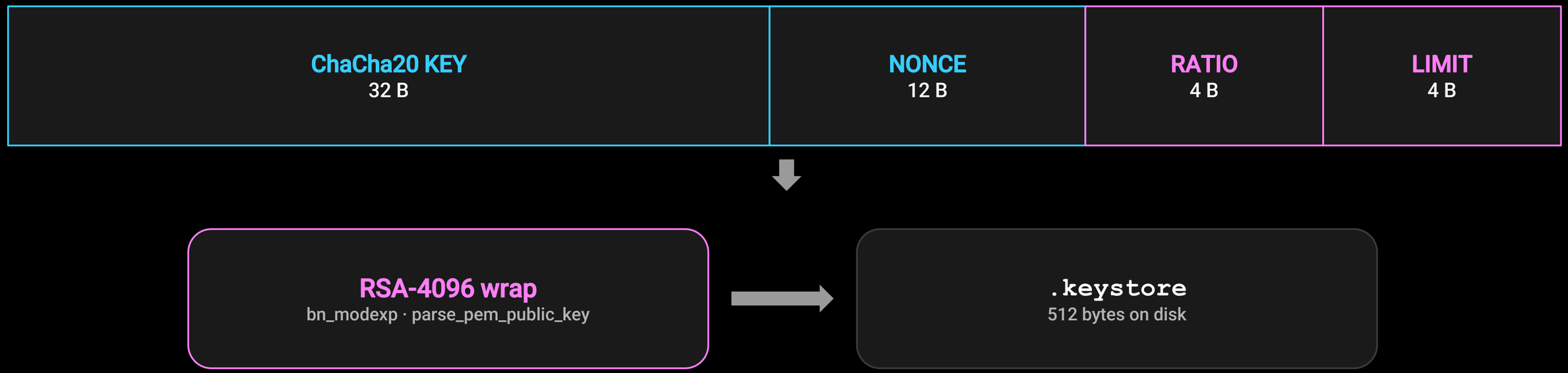
```
__int64 result; // rax
_BYTE v10[64]; // [rsp+30h] [rbp-90h] BYREF
_BYTE v11[48]; // [rsp+70h] [rbp-50h] BYREF
int v12; // [rsp+A0h] [rbp-20h]
int j; // [rsp+B8h] [rbp-8h]
int i; // [rsp+BCh] [rbp-4h]

chacha20_init_state(v11, a1, a2, a3);
for ( i = 0; ; i += 64 )
{
  result = (unsigned int)i;
  if ( i >= a6 )
    break;
  chacha20_block(v11, v10, 20LL);
  ++v12;
  for ( j = i; j <= i + 63 && j < a6; ++j )
    *(_BYTE *)(j + a5) = v10[j - i] ^ *(_BYTE *)(j + a4);
}
return result;
```

Decompiled ChaCha20 keystream generation and XOR loop

THE .keystore — THE KEY, SEALED

Everything needed to decrypt is packed into one 512-byte blob — then sealed with RSA-4096:

PLAINTEXT KEY MATERIAL — 52 bytes



Without the attacker's RSA private key it's meaningless — the lock they trusted. **But the key inside was already broken.**

FLAW #1: PREDICTABLE SEED

THE FLAW

- Key material comes from C's rand()
- Seeded with time(0) — whole seconds
- Deterministic: seed decides the key
- A PRNG, never a CSPRNG

~14,400

candidate seeds in a 4-hour window

File timestamps pin the attack window — brute-forcing the seed is trivial

FLAW #2: srand() INSIDE THE LOOP

The key generator re-seeds every iteration — **so every byte is identical:**

```
unsigned int v2; // eax
__int64 result; // rax
unsigned int i; // [rsp+1Ch] [rbp-4h]

for ( i = 0; ; ++i )
{
  result = i;
  if ( i >= a2 )
    break;
  v2 = time(0LL);
  srand(v2);
  *(_BYTE *)(i + a1) = rand();
}
return result;
```

`time()` is per-second;
the loop runs in microseconds.

Same seed → same `rand()`
→ the SAME byte.

```
critical_generate_srand_rand(v50, 32LL);
critical_generate_srand_rand(v49, 12LL);
critical_generate_srand_rand(v48, 256LL);
```

32-byte key · 12-byte nonce · all from the flawed generator.

256-BIT KEY - 8 BITS OF ENTROPY

The 32-byte key collapses to one repeating byte:

A8	A8	A8	A8	A8	A8	A8	A8	A8	A8	A8	A8	A8	A8	A8	A8	A8

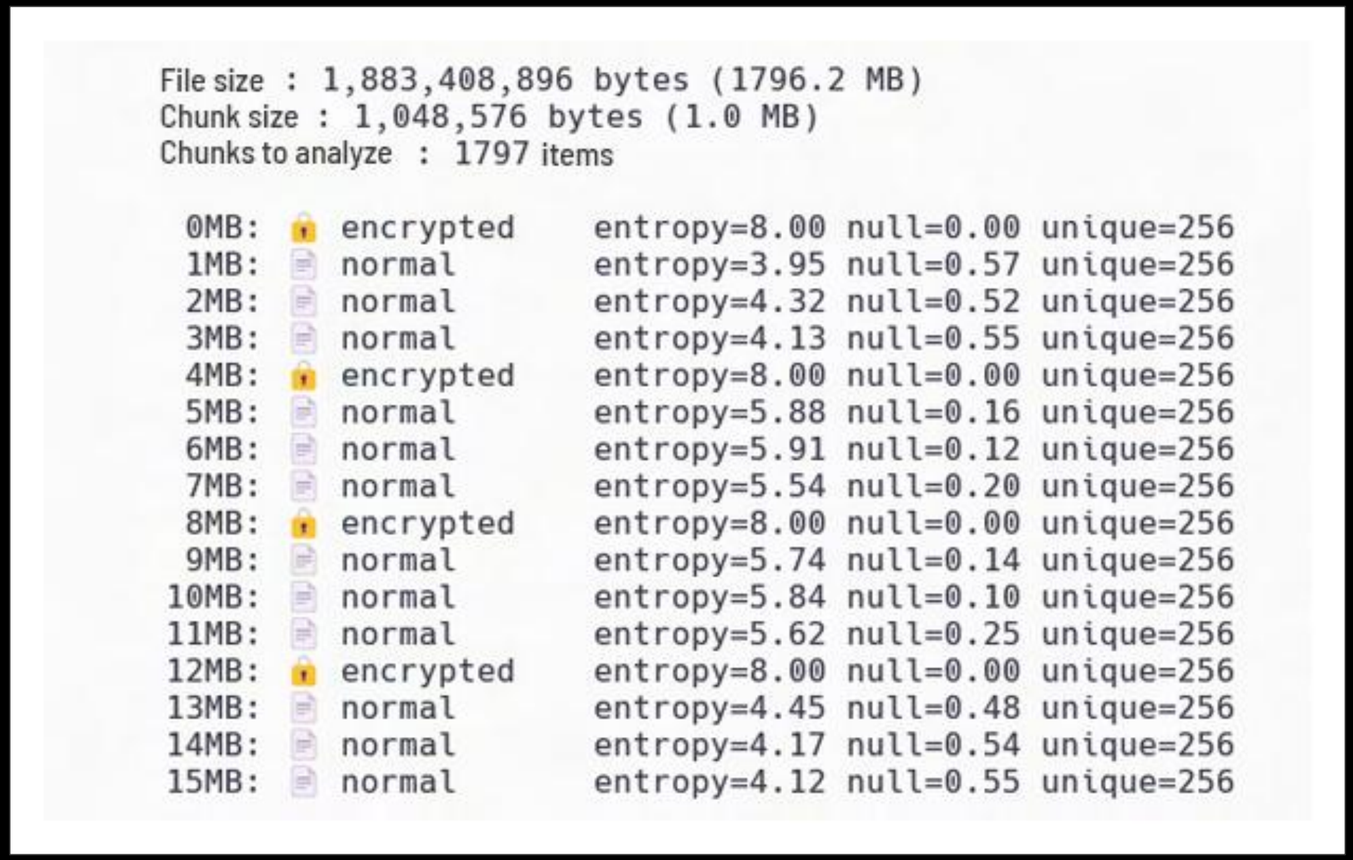
Effective keyspace **2^8 = 256** — guessable in 256 tries. PROOF: two identical files → the same hash

Filename	MD5	SHA1
good.txt	6ab5a529bdfb64ec10e1ef836b02f752	9b37de059f25e0067cbf18a755fa18968a72e
good2.txt	6ab5a529bdfb64ec10e1ef836b02f752	9b37de059f25e0067cbf18a755fa18968a72e

INTERMITTENT ENCRYPTION
1 MB ON - 3 MB OFF

ratio=3 encrypts 1 MB then skips 3 MB.

Block entropy maps the pattern and the recoverable plaintext.



```
File size  : 1,883,408,896 bytes (1796.2 MB)
Chunk size : 1,048,576 bytes (1.0 MB)
Chunks to analyze  : 1797 items

  0MB:  encrypted    entropy=8.00 null=0.00 unique=256
  1MB:  normal       entropy=3.95 null=0.57 unique=256
  2MB:  normal       entropy=4.32 null=0.52 unique=256
  3MB:  normal       entropy=4.13 null=0.55 unique=256
  4MB:  encrypted    entropy=8.00 null=0.00 unique=256
  5MB:  normal       entropy=5.88 null=0.16 unique=256
  6MB:  normal       entropy=5.91 null=0.12 unique=256
  7MB:  normal       entropy=5.54 null=0.20 unique=256
  8MB:  encrypted    entropy=8.00 null=0.00 unique=256
  9MB:  normal       entropy=5.74 null=0.14 unique=256
 10MB:  normal       entropy=5.84 null=0.10 unique=256
 11MB:  normal       entropy=5.62 null=0.25 unique=256
 12MB:  encrypted    entropy=8.00 null=0.00 unique=256
 13MB:  normal       entropy=4.45 null=0.48 unique=256
 14MB:  normal       entropy=4.17 null=0.54 unique=256
 15MB:  normal       entropy=4.12 null=0.55 unique=256
```

Entropy tool — encrypted blocks (8.00) vs normal plaintext, revealing the ratio

DAY 3
BUILDING THE DECRYPTOR

From proof-of-concept to a production recovery engine — LLM-assisted.

D3

LLM-ASSISTED ENGINEERING

AI compressed days of reverse engineering into hours — under human direction:

Human insight + AI-assisted engineering + robust infrastructure, **working in unison**

THE FIRST CRACK: TIMESTAMP KEYS

Before the byte-repeat trick, the very first PoC exploited the predictable seed head-on:

THE PROOF

- Encrypt two identical files → bit-identical output
- Same seed → same key & nonce → same ciphertext
- The seed is a Unix timestamp (seconds)

THE METHOD

- Read file modified-times → a candidate window
- Try each second in range as the srand seed
- Validate: UTF-8 decode + entropy skew < 80%

It worked — but timestamps can be altered and wide windows explode the seed count. **So we went deeper.**

ONE TOOL - EVERY CASE

The PoCs became one hardened decryptor — flexible enough for every scenario:

WHAT WE ENGINEERED

- Merged timestamp & fixed-byte PoCs
- Auto-search the key — or specify it
- Partial-encryption ratio & size limit
- Simulate without overwriting the disk

```
FSI Ransomware Decryptor (User-Guided & Enhanced Key Scan)

positional arguments:
  path                  Path to the file/folder/disk to decrypt

options:
  -h, --help            show this help message and exit
  --type {auto,file,folder,disk}
                        Type of target to process (default: auto)
  --timestamp TIMESTAMP
                        Use a specific timestamp (Unix Epoch, seconds)
  --timestamp-range START_TS END_TS
                        Timestamp scan range (Unix Epoch, seconds)
  --window-hours WINDOW_HOURS
                        Timestamp estimation window (hours, default: 1)
  --timestamp-search    Enable timestamp range scan (File: user-selected, Folder: auto-optimal)
  --fixed-byte FIXED_BYTE
                        Use a specific fixed byte value (e.g., 0xAB or 171)
  --fixed-byte-search   Enable scan based on 256 fixed byte values (0-255) (File: user-selected, Folder: auto-optimal -recommended)
  --ratio RATIO         Partial encryption pattern (MB) - Skip N MB
  --limit LIMIT         Processing limit (GB) - Max data to process
  --block-size BLOCK_SIZE
                        Processing block size (MB) - Data amount to decrypt at once
  --threads THREADS     Number of parallel processing threads (default: 4)
  --extensions EXTENSIONS [EXTENSIONS ...]
                        Encrypted file extensions (default: .ENCRT .encrt .locked .encrypted)
  --dry-run             Perform a test run without actual writes
  --output OUTPUT       Output path (for disk decryption - may not be used in current implementation)
  --verbose, -v         Enable verbose output
```

Decryptor CLI — --timestamp · --fixed-byte · --ratio · --limit · --dry-run · --threads

REBUILDING THE KEYSTREAM

The flaw hands us the key — regenerate the exact keystream and XOR it away:

RECOVER KEY
256 tries

→

REGEN NONCE
same collapse

→

BUILD KEYSTREAM
ChaCha20 per block

→

XOR → PLAINTEXT
cipher ⊕ stream

```
v58 = disk_size;
if ( *(_DWORD *)(v41 + 4108) && v59 < disk_size )
{
  while ( v70 < v59 )
  {
    v58 = v59 - v70;
    v33 = v59 - v70;
    if ( encrypt_data_1MB <= v59 - v70 )
      v33 = encrypt_data_1MB;
    v57 = v33;
    v60 = pread(disk_device_path, read_data_1MB, v33, v70);
    if ( v60 <= 0 )
      break;
    chacha20_xor((__int64)v50, 1u, (__int64)v49, read_data_1MB, v61, v60);
    v56 = pwrite(disk_device_path, v61, v60, v70);
    if ( v56 != v60 )
      break;
    v34 = v58;
    if ( v63 <= v58 )
      v34 = v63;
    v55 = v34;
    v70 += v60 + v34;           // --ratio
    write_progress(v42, v70);
```

The malware's read → ChaCha20 → overwrite → skip loop, reversed.

REVERSING INTERMITTENT ENCRYPTION

Partial encryption isn't only a speed trick — it dictates exactly how the decryptor must run:

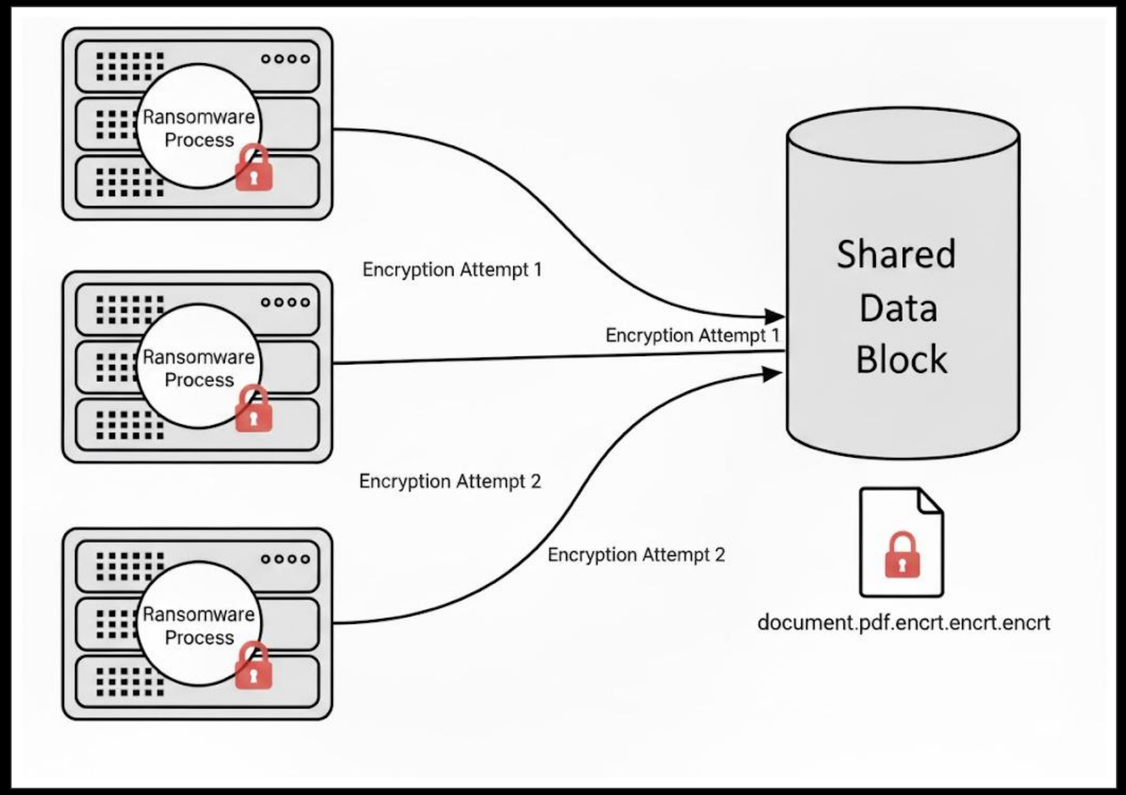
THE GOTCHAS

- Each 1 MB block = its own ChaCha20 counter (start = 1) — not one continuous stream
- Decrypt the encrypted MBs; copy the plaintext MBs untouched
- XOR a plaintext block by mistake → you corrupt it, so ratio & limit must be exact

1 MB decrypt → 3 MB copy, block by block — **the entropy map has to be perfect.**

WHEN ONE KEY WASN'T ENOUGH

Servers sharing one storage volume ran at once
— a single file was encrypted 2–3 times: `.ENCRT.ENCRT.ENCRT`



65,536

double-key combos (256^2)

16,777,216

triple-key combos (256^3)

Each layer needs its own key — so we peel them one at a time

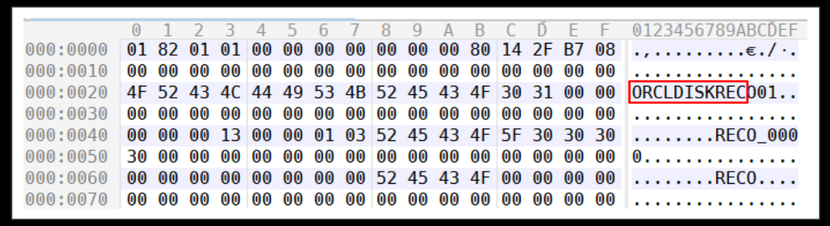
PEELING THE LAYERS

Brute-forcing millions of keys across GB-sized files sounds hopeless — until you cheat:

THE 8-BYTE TRICK

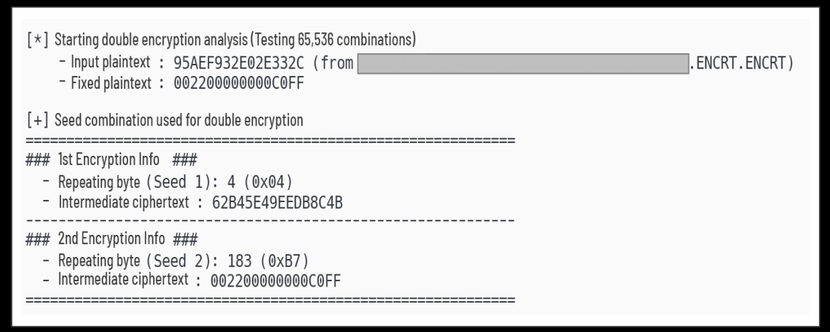
- Sample just 8 bytes at a fixed offset
- Match against a known Oracle header
- ORCLDISKRECO / ORCLDISKDATA signature
- One matching key peels one layer repeat

KNOWN-PLAINTEXT TARGET



```
          0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F  0123456789ABCDEF
000:0000  01 82 01 01 00 00 00 00 00 00 00 80 14 2F B7 08  .,.........€./·.
000:0010  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000:0020  4F 52 43 4C 44 49 53 4B 52 45 43 4F 30 31 00 00  ORCLDISKRECO01..
000:0030  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000:0040  00 00 00 13 00 00 01 03 52 45 43 4F 5F 30 30 30  ........RECO_000
000:0050  30 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  0...............
000:0060  00 00 00 00 00 00 00 00 52 45 43 4F 00 00 00 00  ........RECO....
000:0070  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

DOUBLE-KEY EXTRACTION



```
[*] Starting double encryption analysis (Testing 65,536 combinations)
    - Input plaintext : 95AEF932E02E332C (from [redacted].ENCRT.ENCRT)
    - Fixed plaintext : 002200000000C0FF

[+] Seed combination used for double encryption
============================================================
### 1st Encryption Info  ###
  - Repeating byte (Seed 1): 4 (0x04)
  - Intermediate ciphertext : 62B45E49EEDB8C4B
------------------------------------------------------------
### 2nd Encryption Info ###
  - Repeating byte (Seed 2): 183 (0xB7)
  - Intermediate ciphertext : 002200000000C0FF
============================================================
```

8 bytes instead of a whole GB → **double keys in seconds, triple in ~20 minutes**

IS THIS THE RIGHT KEY?

Testing 256 candidate bytes is trivial — knowing which one worked is the hard part. Each sample must pass:

THE VALIDATION PIPELINE

- Entropy gate — sample entropy < 7.5
- Magic bytes — PNG / PDF / ZIP / Oracle
- ASCII ratio — human-readable text %
- Byte diversity — null ratio & unique count
- UTF-8 decode — no decode error

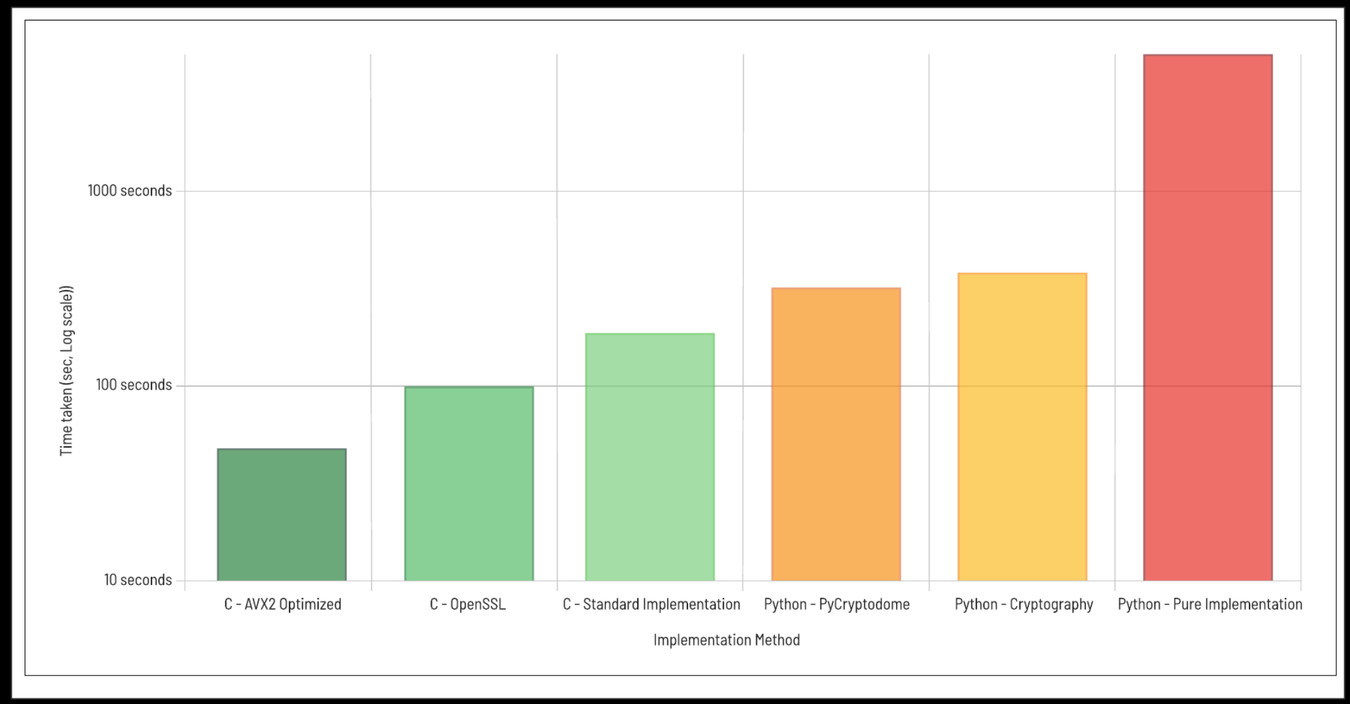
WHY IT MATTERS

- Wrong key → still high-entropy noise
- XOR on plaintext would corrupt it
- So the ratio & limit must match too
- Score > threshold = the real key

256 tries, auto-confirmed — **no human eyeballing gigabytes of hex.**

ENGINEERING A 320× SPEEDUP

A **SIMD / AVX-optimized** C engine — 100 GB that took 16 h in Python now decrypts in 147 s



Time to decrypt 100 GB (log scale, lower = faster). AVX2-optimized C vs pure Python

WHY C WON — THE QUARTER-ROUND

ChaCha20's core is a 32-bit rotation, run hundreds of millions of times — that's where Python dies:

THE 32-BIT ROTATION

```
(x << n) | (x >> (32 - n))
```

- C: fixed 32-bit int → one ROL CPU instruction
- Python: bigint → manual overflow + typing overhead

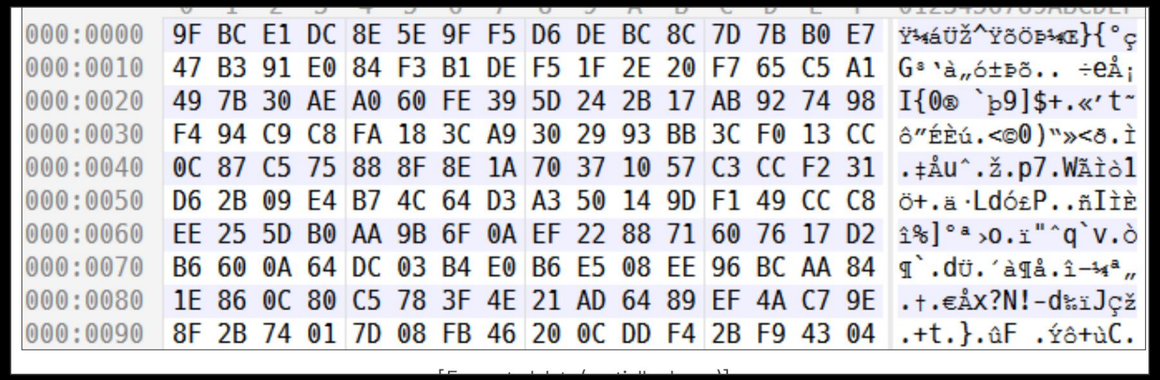
THROUGHPUT — C VS PYTHON

- Optimized C: 355–366 MiB/s (1 core)
- SIMD AVX2: up to 2.19 GB/s
- PyCryptodome: 70–85% of C
- Pure Python: 1–5% of C (GIL blocks threads)

Port the verified logic to C + SIMD → **100 GB from 16 h to 147 s.**

PROOF: THE DATA CAME BACK

ENCRYPTED — random bytes



	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0123456789ABCDEF
000:0000	9F	BC	E1	DC	8E	5E	9F	F5	D6	DE	BC	8C	7D	7B	B0	E7	Ÿ¼áÜŽ^Ÿõ…
000:0010	47	B3	91	E0	84	F3	B1	DE	F5	1F	2E	20	F7	65	C5	A1	G³'à„ó±Þõ.. ÷eÅ¡
000:0020	49	7B	30	AE	A0	60	FE	39	5D	24	2B	17	AB	92	74	98	I{0® `þ9]$+.«'t˜
000:0030	F4	94	C9	C8	FA	18	3C	A9	30	29	93	BB	3C	F0	13	CC	ô"ÉÈú.<©0)"»<ð.Ì
000:0040	0C	87	C5	75	88	8F	8E	1A	70	37	10	57	C3	CC	F2	31	.‡Åu^.Ž.p7.WÃÌò1
000:0050	D6	2B	09	E4	B7	4C	64	D3	A3	50	14	9D	F1	49	CC	C8	Ö+.ä·LdÓ£P..ñIÌÈ
000:0060	EE	25	5D	B0	AA	9B	6F	0A	EF	22	88	71	60	76	17	D2	î%]°ª›o.ï"ˆq`v.Ò
000:0070	B6	60	0A	64	DC	03	B4	E0	B6	E5	08	EE	96	BC	AA	84	¶`.dÜ.´à¶å.î–¼ª„
000:0080	1E	86	0C	80	C5	78	3F	4E	21	AD	64	89	EF	4A	C7	9E	.†.€Åx?N!-d‰ïJÇž
000:0090	8F	2B	74	01	7D	08	FB	46	20	0C	DD	F4	2B	F9	43	04	.+t.}.ûF .Ýô+ùC.

RECOVERED — ORCLDISK signature

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0123456789ABCDEF
000:0000	01	82	01	01	00	00	00	00	00	00	00	80	14	2F	B7	08	.‚.....€./·.
000:0010	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
000:0020	4F	52	43	4C	44	49	53	4B	52	45	43	4F	30	31	00	00	ORCLDISKRECO01..
000:0030	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
000:0040	00	00	00	13	00	00	01	03	52	45	43	4F	5F	30	30	30	........RECO_000
000:0050	30	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	0.....
000:0060	00	00	00	00	00	00	00	00	52	45	43	4F	00	00	00	00	........RECO....
000:0070	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
000:0080	00	00	00	00	00	00	00	00	52	45	43	4F	5F	30	30	30	........RECO_000
000:0090	30	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	0

The decrypted disk block reveals the Oracle `ORCLDISK RECO` header —
a real, mountable database again.

DAY 4 RECOVERY

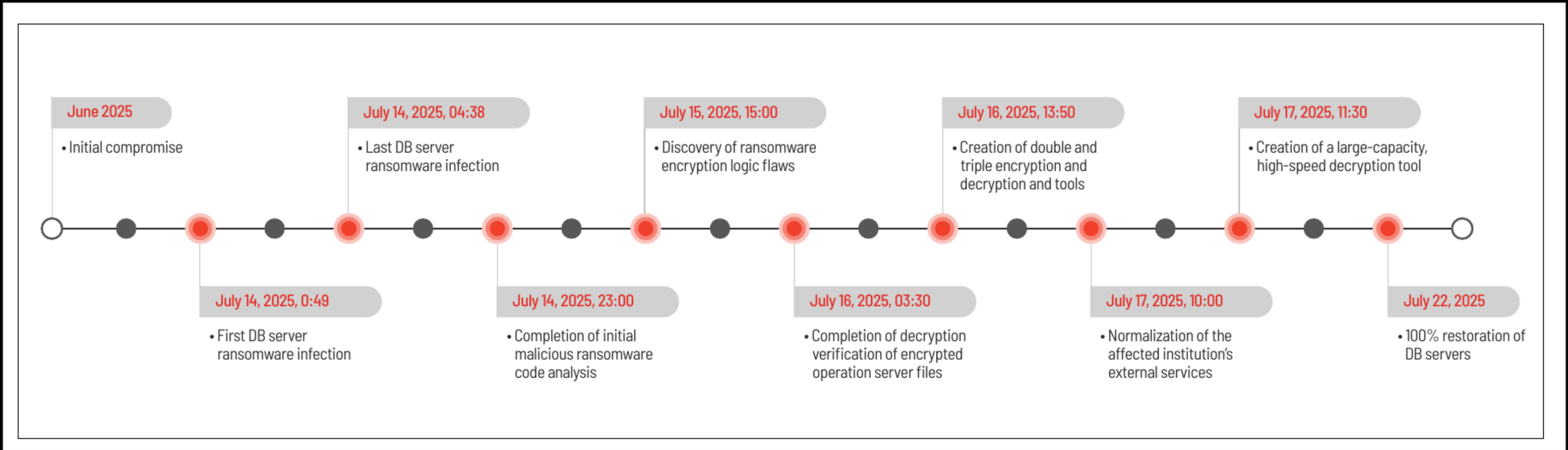
Major services back in 81 hours
— every DB server restored.

D4

100% RESTORED IN 81 HOURS

All DB servers recovered

Public services resumed **81 hours** after the attack — full restoration days later



THE 26-HOUR PIVOT

Why did a sophisticated crew ship broken crypto? They changed their whole toolkit at the last minute.

2025.07.12 19:15 KST
Legacy build
bespoke, per-server toolsets

2025.07.13 21:49 KST
Unified binary
the pivot — 26 h after the legacy build

2025.07.14 00:49 KST
Execution
deployed at once — zero QA time

A last-minute shift to a single unified routine — zero time for QA — **introduced the flaw we exploited**

WHY THE FLAW EXISTED

LEGACY (SECURE)

- 40+ bespoke per-server binaries
- 40+ unique RSA key pairs
- Stable — high operational overhead
- No shared crypto weakness

26h pivot

UNIFIED (FLAWED)

- One "one-size-fits-all" binary
- Compiled 26 h before the attack
- Zero QA before deployment
- Shared srand-in-loop flaw

40+ target-specific builds collapsed into one hasty binary — **the root of failure.**

IMPLICATIONS FOR DEFENDERS

SERVER-INFILTRATION

- Hands-on, not spray-and-pray
- Custom scripts per target
- Linux / Oracle DB in the crosshairs

DOUBLE EXTORTION

- Exfiltrate before encrypting
- Name-and-shame data-leak sites
- Pressure beyond the lockout

RESILIENCE

- Harden the attack surface
- Offline, immutable backups
- Regular restore drills

Detection matters — but recovery is engineered before the attack, **and proven after it**

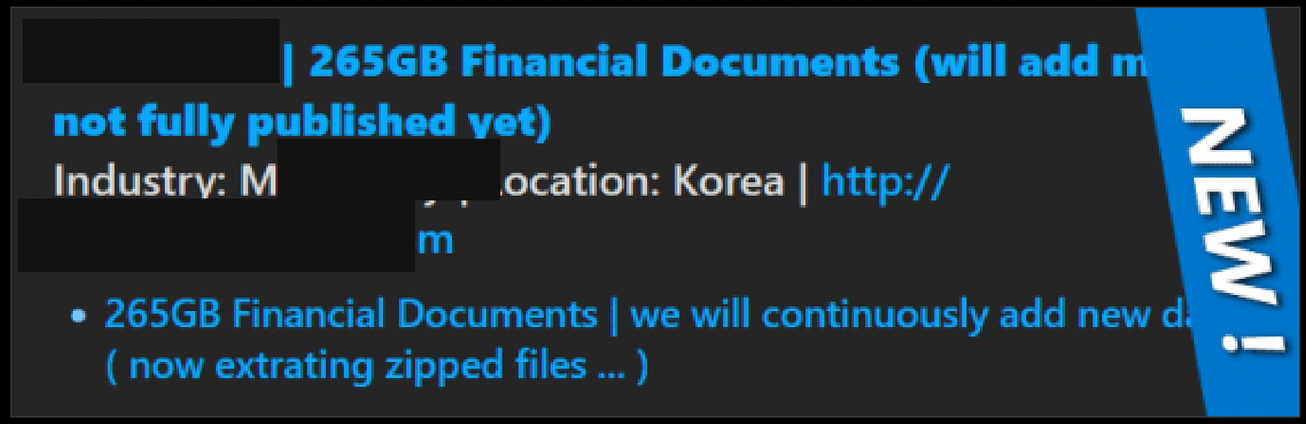
GUNRA HASN'T STOPPED

The same group stays active — tracked on their C2 and their data-leak site

DOUBLE EXTORTION

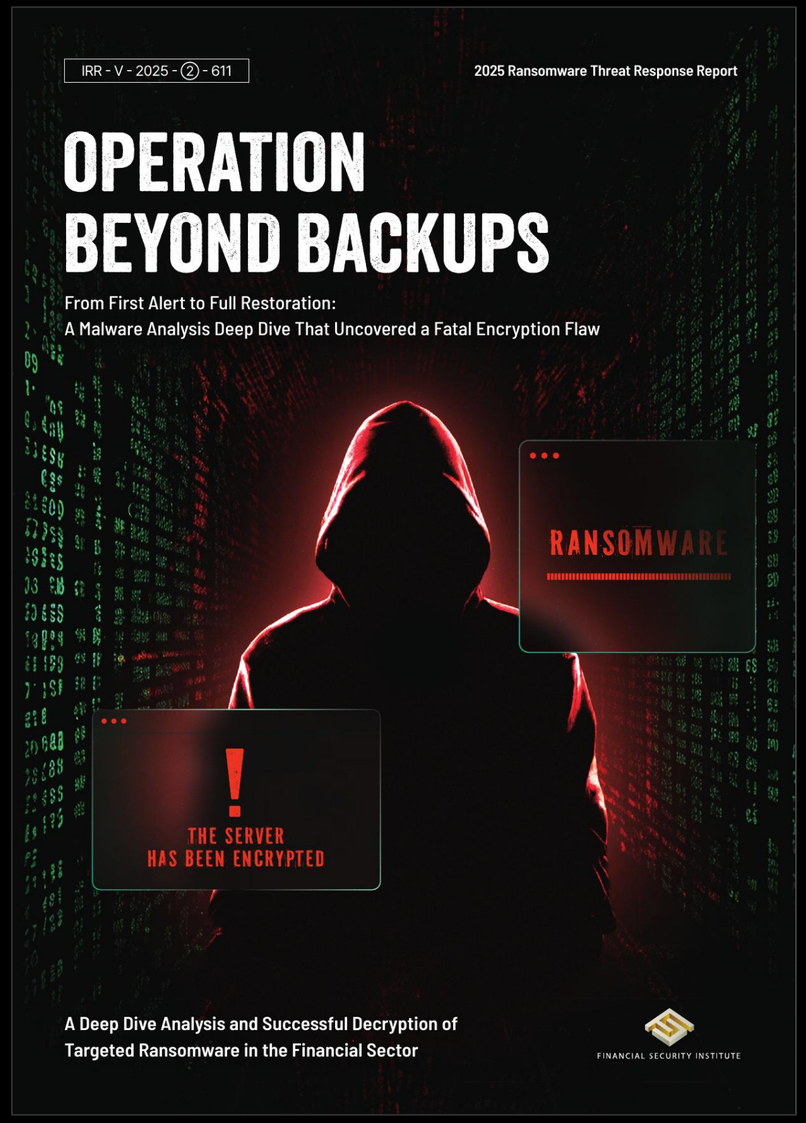
- Exfiltrate data before encrypting
- Name-and-shame on a public leak site
- Pay — or the stolen data is published

GUNRA DATA-LEAK SITE



Next-wave tooling staged on C2: `Sliver RAT (linux)` · `OpenSSH-Win64` — **monitoring continues**

THE FULL PLAYBOOK



Operation Beyond Backups

2025 Ransomware Threat Response Report

Financial Security Institute — CERT · Nov 2025

- End-to-end incident-response lifecycle
- Malware teardown & cryptographic audit
- The decryption-engine architecture
- **A No-Ransom blueprint for defenders**

REDEFINING RANSOMWARE RESPONSE THROUGH TECHNICAL RIGOR

The Evolutionary Threat

hands-on, double-extortion ransomware is an engineering challenge, not a lock.

- **The Power of Depth —** even "impossible" attacks hide implementation flaws.
- **Resilience is a Choice —** proactive ASM + immutable, tested backups.

The No-Ransom Blueprint

human insight + AI-assisted engineering + robust infra, in unison.

THANK YOU

Financial Security Institute — Code Analysis Team

SungWook Jang · YoungMook Kang
DaeGyu Kang · Younghwan Kim · Ahyun Song

swjang@fsec.or.kr · #BHUSA @BlackHatEvents
kangyoungmook@fsec.or.kr kr · #BHUSA @BlackHatEvents