Akira Ransomware: In-Depth Technical Analysis

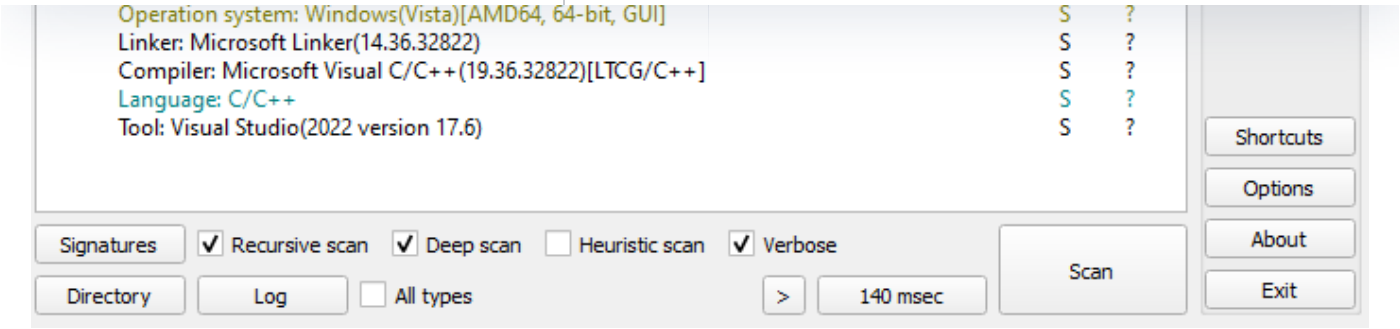
CASE STUDY RANSOMWARE

Akira's current iteration exemplifies a high level of technical sophistication, combining advanced cryptographic techniques with complex multithreaded operations. Developed in **C++** and utilizing the **Boost.Asio** library for asynchronous multithreading, Akira efficiently manages concurrent encryption tasks, enhancing its performance and complicating reverse engineering efforts. The ransomware employs a hybrid encryption scheme that integrates **KCipher2** and **ChaCha8** stream ciphers for data encryption, alongside **RSA-4096** with **PKCS#1 v1.5** padding for encrypting the keys/nonces. This multifaceted approach ac layers of complexity.

depth reverse-engineering will reveal these mechanisms in detail. In doing so, it aims to translate Akira's complex internals into practical insights for defenders, illustrating why analyzing this malware is essential for staying ahead of modern ransomware threats.

Technical analysis

The ransomware is written in C++ and makes use of **Boost ASIO** for multithreading, **spdlog** for logging, **ADVobfuscator** for string encryption (only a couple of critical strings are encrypted), and **Nettle** (cryptographic library) for crypto. The sample we are analyzing is a Windows sample for the new version of Akira, and it was compiled on **2025-03-19 22:53:17**. So far, all the samples that we have seen for the new version of Akira have had the same compilation time, which suggests that the victim-specific information is patched into a precompiled binary using a builder.



Strings Set Initialization

Before calling the **WinMain** function, five custom functions are called via **_initterm** (an internal method that walks a table of function pointers and initializes them). In three of these functions, strings are decrypted at runtime; while in the other two functions, the strings are in plaintext. The strings are encrypted with **AdvObfuscator**, and a different set is created in each of these five functions. The following sets are created: **fullEncryptExts**, **VMExts**, **ignoredProcesses**, **IgnoredExtensions**, and **IgnoredFolders**. A complete list for each of these sets can be found at the end of this blog post.

For each extension in **fullEncryptExts**, the file is 100% encrypted regardless of the specified

The **ignoredProcesses** set is used to store the process IDs of critical processes. These **PIDs** are later ignored when Akira fails to obtain a handle to a file and attempts to terminate any programs using that file. **WTSEnumerateProcessesW** is used to retrieve information about the active processes.

```
        p_ws20 = &p266->ws20;
        if...
        p_processName1 = &processName1;
        v5 = processName1.size;
        if...
        if...
        if...
        v14 = 1;
LABEL_22:
        if...
        if ( v14 )
        {
          pid = &pMemory[v2].ProcessId;
          if ( *(&ignoreProcIds + 1) == *(&ignoreProcIds + 2) )       // proc id vector
          {
            sub_140071400(v5, *(&ignoreProcIds + 1), pid);
          }
          else
          {
            *(_DWORD *)*(&ignoreProcIds + 1) = *pid;
            *(&ignoreProcIds + 1) = (char *)*(&ignoreProcIds + 1) + 4;
```

WinMain Function

The ransomware first creates a log file in the same folder which contains information about the execution. The format of the filename is **Log-day-month-year-hour-minute-second.txt.** For example : Log-01-05-2025-22-30-20.txt. The date is in local time. All the errors, and other info is logged in this file.

```
getCurrentTime(&Time);
localtime = localtime64(&Time);
strftime(localtimeString, 0x50uLL, "Log-%d-%m-%Y-%H-%M-%S", localtime);
memset(&v181, 0, sizeof(v181));
v5 = 0xFFFFFFFFFFFFFFFFuLL;
do
  ++v5;
while ( localtimeString[v5] );
string::assign(&v181, localtimeString, v5);
logger::init(v6, &v181);
```

Here is how the logged data looks like :

Akira supports the following command-line arguments:

CLI Args	Definition
–exclude / -e	Exclude files whose filename matches the provided regex.
–encryption_path / -p	Specifies the path for the directory where files will be recursively encrypted.
-localonly	Only encrypt the local drives. (Remote drives are excluded)
-l	Log the available drives in the log file. (Nothing is encrypted)
–share_file / -s	A file which contains paths to recursively encrypt. The file must have each path separated by a new line.
–encryption_percent / -n	The encryption percentage (integer) used by the ransomware.
-dellog	Clear event logs using PowerShell.

The arguments are passed in a non-standard way. For example, '**–encryption_percen**', an equal sign is used to specify the argument. If no encryption path or share file is pr...d, the ransomware will recursively encrypt the data in each drive.

If the **-dellog** argument is provided, akira proceeds to run the following command to clear event logs.

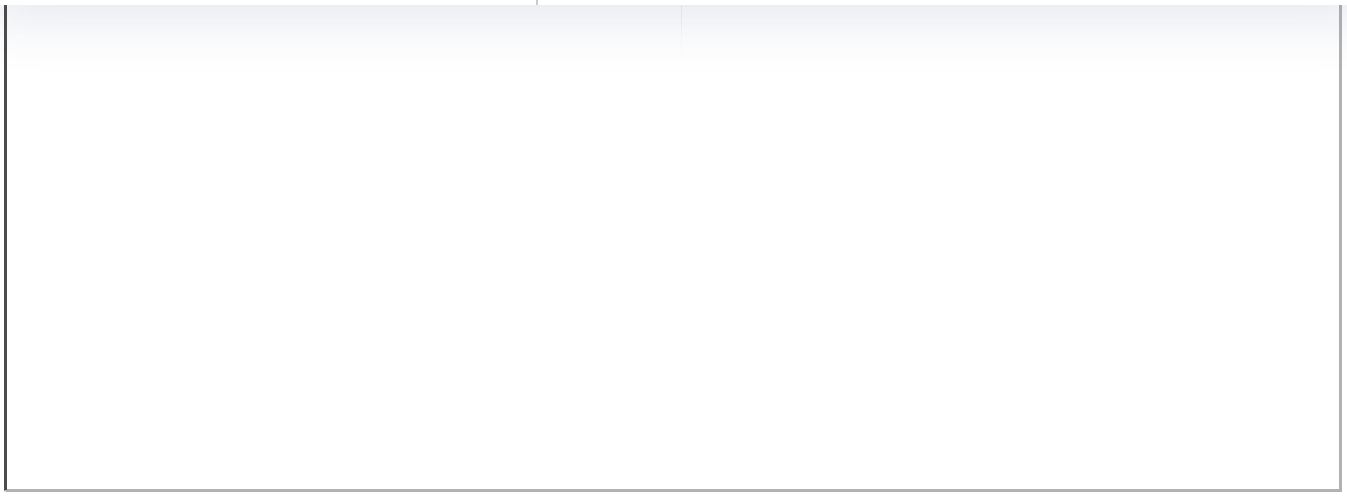
```
 powershell.exe -ep bypass -Command Get-WinEvent -ListLog * | where { $_.RecordCount } | ForEach-Object -Process{ [System.Diagnostics.Eventing.Reader.EventLogSession]::GlobalSession.ClearLog($_.LogName) }
```

This command first bypasses the execution policy and then runs a command that clears each event log.

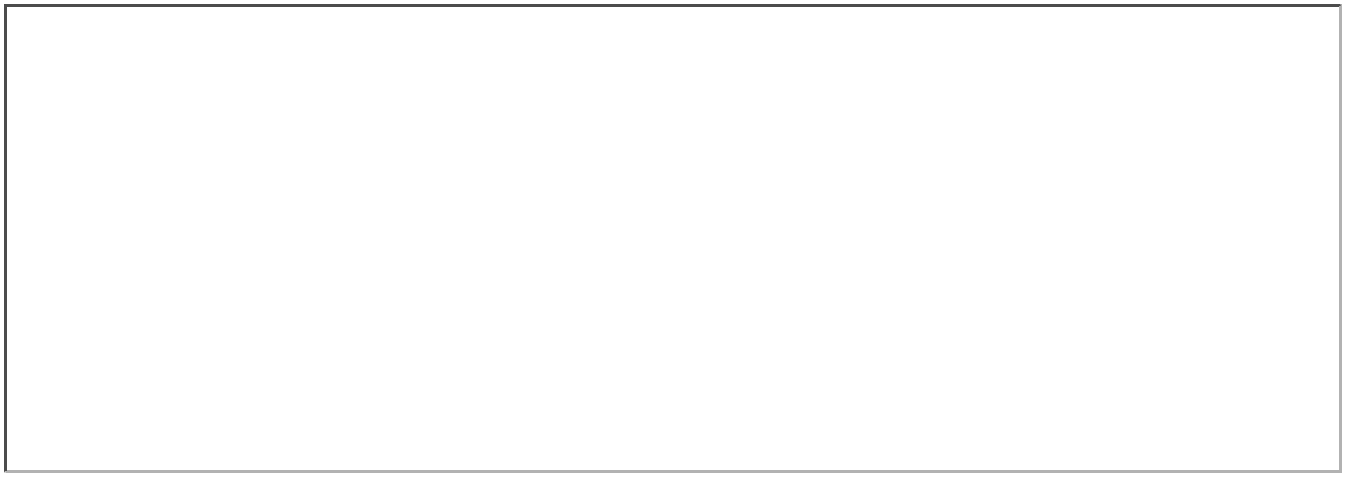
Next, the ransomware deletes the **shadow copies** using the following command, the command is decrypted at runtime.

```
“powershell.exe -Command "Get-WmiObject Win32_Shadowcopy | Remove-WmiObject"”
```

The command is executed using **Windows COM** and **WMI** to launch a new process through



If a non-zero **process ID** is returned, and a handle is obtained with OpenProcess, it waits 15 seconds to ensure that the command has finished executing.



It then uses **GetSystemInfo** to fetch the number of processors, and then it initializes the public key in the **CryptoCtx**.

initialized (only seeded). It's later used as the random function for **PKCS#1 v1.5** padding. The seed for this function is generated the same way as in the old vulnerable version of akira, but the seeding part was changed for the key/IV generation. Therefore, it still remains vulnerable here; however, that's not very useful as this is only used for **PKCS#1 v1.5** padding. We'll discuss the specifics of the new seeding scheme and the old seeding scheme later on in the blog.

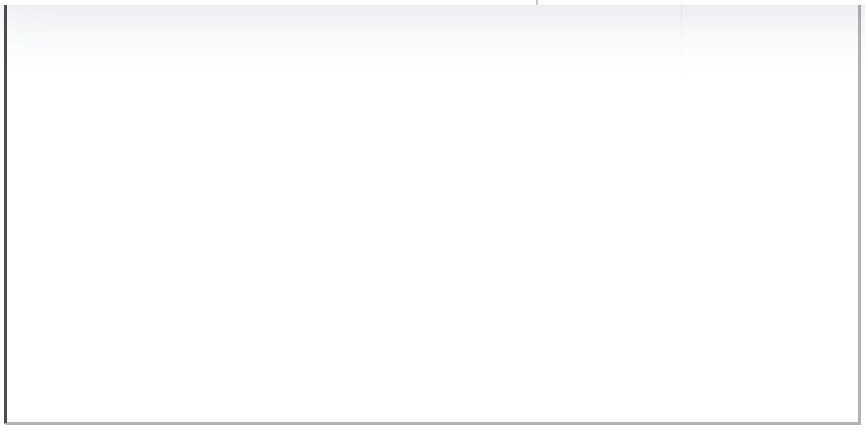
The number of processors obtained is set to two if there is only a single processor, then multiplied by two to calculate the total thread count. A minimum of four threads is guaranteed. Next, the numbers of threads dedicated to root-folder parsing, folder parsing, and encryption are calculated, and the corresponding thread pools are initialized using Boost.Asio. Sixty percent of the threads are allocated to encryption, thirty percent to folder parsing, and ten percent to root-folder parsing. There will always be at least two threads for encryption and one thread each for folder and root-folder parsing.

The folder and root-folder parser functions skip any files whose extensions appear in **IgnoredExtensions**, as well as any folders whose names appear in **IgnoredFolders**. For each directory encountered, a function writes the ransom note in that directory, the name of the ransom note is "**akira_readme.txt**". The login code referenced in the ransom note is also encrypted and embedded both in each encrypted file, and the **.arika** files.

Get Handle To File

For each file to be encrypted, the ransomware first clears the "**read-only**" attribute if it is set. It then attempts to open the file using the **GENERIC_READ**, **GENERIC_WRITE**, and **DELETE** access masks.

If it fails to open the file, it will kill any processes that are using the file and attempt to get a handle again. Akira uses the **Restart Manager APIs** to kill any processes using the file. **RmStartSession** is used to start a Restart Manager session, **RmRegisterResources** is used to register the file, **RmGetList** is used to get a list of all the processes that are currently using the file, and **RmShutDown** to kill the processes. The process ID for each process is checked against each pid in the **ignoreProcIds** vector, which was previously initialized with the process IDs of critical processes, and if a match is found, that process is skipped. **RmShutDown** is used to kill the processes, and at the end, **RmEndSession** is used to end the Restart Manager session.



Once the ransomware gets a handle to the file, it will get the file size using **GetFileSizeEx**, and then it will decide the encryption mode depending on the file size and the file type. For each file, an auto-save file is generated (**.arika** file). This file contains data required to decrypt the file in case encryption is interrupted. This file is not created when the file size is smaller than 2MB, and the file extension is not in the **fullEncryptExts/VMExts** set. It will then generate the keys/IVs for **ChaCha8** and **KCipher2**.

FileType details: 0 for full encryption; 1 if the file is ≤ 2,000,000 bytes (2MB) and not a VM file; 2 if the file is > 2,000,000 bytes (2MB) or if the file is a VM file.

Keys and Nonces/IVs Generation

Akira ransomware uses both **KCipher2**, and **ChaCha8**. It generates two keys, and two nonces/ivs for each file.

The **GenerateRandomBytes** function uses the **xtime_get_ticks** function to get the system time in 100-nanosecond intervals since the epoch. The **xtime_get_ticks** function uses **GetSystemTimePreciseAsFileTime** if available, otherwise it will use **GetSystemTimeAsFileTime**. This function is called 65 times within **GenerateRando es**, and it's used to generate a long string of 64 numbers. The initially returned ticks are XORed

```
  xoredTicks = 100 * ticks;
i = 0;
while ( 1 )                                     // 64 iterations
{
  xoredTicks_1 = xoredTicks ^ (100 * Xtime_get_ticks());
  v11 = v26;
  if ( xoredTicks_1 >= 0 )                      // convert the 64 bit int to a string
  {
    do
    {
      *--v11 = xoredTicks_1 % 0xAuLL + 48;
      xoredTicks_1 /= 0xAuLL;
    }
    while ( xoredTicks_1 );
  }
  else
  {
    v12 = -xoredTicks_1;
    do
    {
      *--v11 = v12 % 0xA + 48;
      v12 /= 0xAuLL;
    }
    while ( v12 );
    *--v11 = 45;
  }
  __wind
  {
    memset(&numberStr, 0, sizeof(numberStr));
    if ( v11 == v26 )
```

After the loop ends, the resulting string of numbers seeds a **Yarrow256** CSPRNG to generate the random bytes. This implementation uses **Nettle's Yarrow256**, which first performs 1500 rounds of **SHA-256** during the **yarrow256_seed** call and derives an initial counter value by encrypting a null counter with **AES-256**. Subsequently, **yarrow256_random** generates random bytes by encrypting the counter with **AES-256** and incrementing it as a big-endian number for every 16 bytes produced. These encrypted counter values become the final random bytes.

The **GenerateRandomVal** function uses **GenerateRandomBytes** to generate the random value, and the random bytes are xored with the previous randomValue if it's not null.

After this, the endianness is swapped for the keys and nonces, and the keys, and nonces are split up before they are written to a new struct (plaintext struct); this doesn't affect the original keys, and nonces. The struct is then encrypted with **rsa_encrypt** from **Nettle**. **Yarrow256** is used as the random function for **PKCS#1 v1.5** padding.

Here is the structure for the plaintext:

```
#pragma pack(push, 1)
struct plaintext
{
  unsigned __int8 constant_one;
  unsigned __int8 filetype; // 0 for full encryption; 1 if the file
is ≤ 2,000,000 bytes (2mb) and not a VM file; 2 if the file is >
2,000,000 bytes (2mb) or if the file is a VM file.
  unsigned __int8 enc_percent;
  unsigned __int64 file_size;
  unsigned __int8 ChaCha8_key_1[8];
  unsigned __int8 ChaCha8_nonce_1[4];
  unsigned __int8 ChaCha8_key_2[8];
  unsigned __int8 ChaCha8_nonce_2[4];
  unsigned __int8 ChaCha8_key_3[8];
  unsigned __int8 ChaCha8_nonce_3[4];
  unsigned __int8 ChaCha8_key_4[8];
  unsigned __int8 ChaCha8_nonce_4[4];
  unsigned __int8 KCipher2_key_1[4];
  unsigned __int8 KCipher2_nonce_1[4];
  unsigned __int8 KCipher2_key_2[4];
  unsigned __int8 KCipher2_nonce_2[4];
```

instructions. Then **ChaCha8** and **KCipher2** are initialized with the previously derived keys and IVs/nonces.

Taking a look at the old vulnerable version

The previous version of Akira contained a fatal flaw in its **Yarrow256** seed generation. It relied on **QueryPerformanceCounter**, and the frequency returned by **QueryPerformanceFrequency** to create a seed. On modern Windows systems the frequency is typically fixed at ten million ticks per second, while on older systems it hovers around two to three million ticks per second (in this case, the frequency changes only slightly between boots) or in some cases remains constant at **14 318 180** ticks. The counter value is reset at boot. As a result, it was possible to brute force the counter value and decrypt data without

Two counter values are required for each cipher because the nonce is generated almost immediately after the key. We'll call the counter at key generation **tick1** and the counter at nonce generation **tick2** for a given cipher. To brute-force **tick1**, we first need the system's most recent boot time – recorded as **Event ID 12** in the Windows system event log. We also need a reasonable lower bound for when the ransomware encrypted our target file, which we can derive from the creation timestamp of a reference file (for example, the log file or the ransom note) or from timestamps inside the log itself; the PowerShell event for deleting shadow copies provides a similarly precise marker for the time the ransomware started. Subtracting the boot time from this "start" timestamp gives the number of seconds elapsed since boot. Multiplying that by the frequency produces the approximate starting counter value for our search.

Next, we can patch inline hooks into the ransomware's **GenerateRandomBytes()** calls to log the actual tick counts. From these logs, we can determine the typical range of values for tick2 – tick1. With that range in hand, we brute-force both tick1 and tick2 at full timer resolution. For each candidate second (covering every tick within that second, where each tick represents a possible tick1), we compute the corresponding tick2 by brute-forcing values within the typical tick2 – tick1 range and adding those to tick1. As the brute-force progresses, we continue incrementing the candidate's second until we achieve a successful crack.

version, multiple RTX 4090s can break a key/nonce pair for a VM's flat disk or snapshot in a few days. We have helped several clients restore their critical VMs and databases because of this critical vulnerability in Akira. The vulnerability was first publicly discussed by **Fitsec Ltd**. Later on, in 2025, a researcher published their own decryptor for the ESXi version of Akira, and as a result, the vulnerability was patched in the latest version of Akira only a few days later.

Is the latest version vulnerable?

The latest Akira build dramatically increases the complexity of seed generation by using dozens of high-precision timestamps instead of a single performance-counter value. To recover a key and nonce, you would now have to brute-force the precise time offsets for each of the roughly 130 calls to **xtime_get_ticks** in **GenerateRandomBytes** during key/nonce setup—on top of the additional 65 calls made when **GenerateRandomVal** is called (once in main, once after encrypting each page, and four times in the key/nonce-generation function). The timing of each **GenerateRandomVal** call is also hard to predict, since it occurs after encrypting each page in a file as well.

Moreover, real-world factors introduce huge variability in those offsets. Context switches can abruptly delay individual ticks, and heap reallocations for expanding the seed buffer can add unpredictable pauses. On top of that, **randomValue** itself is generated via **GenerateRandomBytes** inside **GenerateRandomVal** and is shared across threads as it's a globalVariable, so an attacker would need to predict every single **xtime_get_ticks** call in the under multithreaded conditions. Together, these factors ensure that the current seed generation scheme cannot be broken without a future implementation mistake by the at actor.

with this name already exists. If it does, the current file is skipped, and the ransomware proceeds with the next file.

If a **.arika** file is created, later on in the encryption function after the ransomware encrypts each page of the file (page size of **0xFFFF**), it will update the **.arika** file with the new max enc offset. Additionally, the .**arika** file also holds the login code from the ransom note. The code is right-padded with null bytes to a length of 0x50, then encrypted with **KCipher2**. The encryption key is the first 16 bytes of the MD5 hash (string) of the filename, and the IV is the hash's last 16 bytes.

```
if ( !kcipher2::init(&cipherCtx, 16LL, key, 16LL, iv) )
{
  v5 = operator new(0x50uLL);
  buffera[0] = v5;
  v10 = v5 + 80;
  *v5 = 0LL;
  *(v5 + 1) = 0LL;
  *(v5 + 2) = 0LL;
  *(v5 + 3) = 0LL;
  *(v5 + 4) = 0LL;
  buffera[1] = v5 + 80;
  v6 = buffera[0];
  *buffera[0] = *code;                    // code / id (mentioned in the ransomnote).
                                          // right padded with null bytes
  v6[1] = *&code[2];
  v6[2] = *&code[4];
  v6[3] = *&code[6];
  v6[4] = *&code[8];
  if ( kcipher2::encrypt(&cipherCtx, 0x50LL, buffera[0]) )
```

Here is the complete structure of the .arika file:

```
struct arika{
unsigned char encryptionState; //set to 0xFF initially. It is later
set to 0x0 after a page is encrypted and is about to be written to
the file. After the page is written to the file, it is set to 0x
unsigned int64 max_enc_offset; // The max offset akira was able to
```

File Encryption

Based on the file type, workers are initialized to perform encryption.

encryptLarge performs intermittent encryption, **encryptSmall** performs partial encryption (in the header), and **encryptFull** completely encrypts the files.

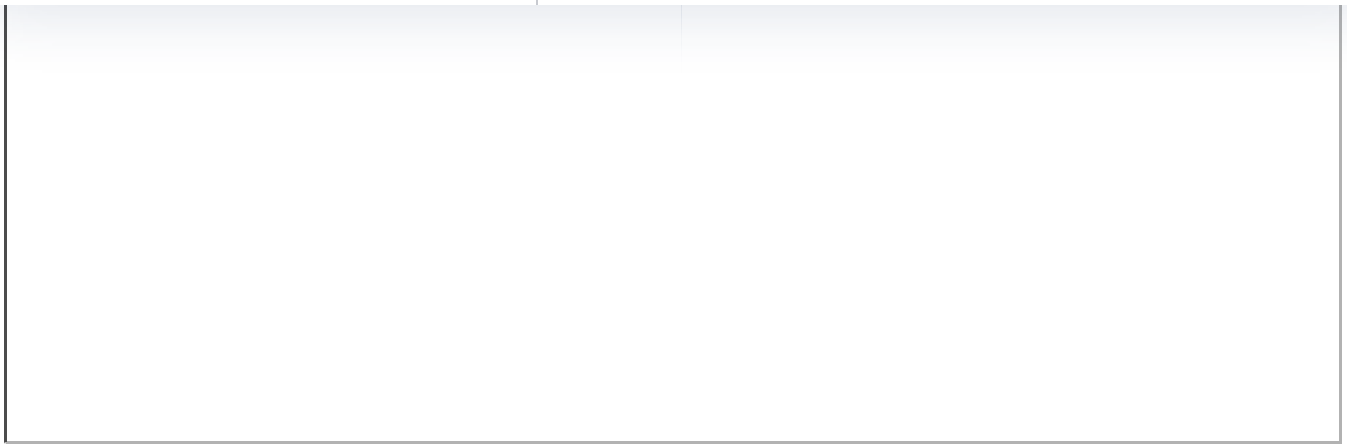
encryptLarge uses the algorithm shared below to calculate the chunk gap (gap between the start of one encrypted chunk and another encrypted chunk), and the encrypted block size. If the encryption percentage is higher than or equal to **50%**, the encrypted block count is set to 4; otherwise, the encrypted block count is set to 2.

If the encryption percentage exceeds **57** percent, the **chunkGap** becomes slightly smaller than the **encryptedBlockSize**. As a result, each new block encryption will re-encrypt a portion of the previous block – and increasing the percentage further only shrinks the **chunkGap** even more. For instance, with a **33 554 432-byte** file and a **100** percent encryption percentage, the **chunkGap** is **0x147AE1** while the encryptedBlockSize is **0x666666**, so each subsequent block encryption overlaps and re-encrypts a portion of the data from the blocks before it.

Here is a Python function to calculate these values:

```
def compute(file_size, enc_percentage):
    if enc_percentage < 0x32:
        n5 = 3
        enc_block_count = 2
    else:
        n5 = 5
        enc_block_count = 4
    enc_block_size = (file_size * enc_percentage) // 0x64 // n5
    chunk_gap = (file_size - enc_block_size * enc_block_count) // n5
    print(f"encBlockSize : {hex(enc_block_size)}")
    print(f"chunkGap : {hex(chunk_gap)}")
```

The **encryptSmall** function encrypts a single contiguous block starting at the file's b ng. whose size is equal to encryptPercentage percent of the filesize**.** For example, for a fi ose size is **0x150000** and the encryption percentage is **75%**, a block of **0xFC000** (**75%** of



Every first page and last page in a block is encrypted with **KCipher2**, while all the other pages are encrypted with **ChaCha8**.

Once the page is encrypted, the **max_enc_offset** is updated in the **.arika** file, and the **encryptionState** byte in the arika file is set to 0, after which the encrypted data is wr... to the file. The **encryptionState** byte is set back to 1 once the encrypted data has been ...en to the file. After this, Akira keeps repeating the same process until the file is completely

After a file has been fully **encrypted**, the encrypted login code and the encrypted plaintext structure are appended to its end, and Akira then proceeds to the next file.

Conclusion

The technical analysis presented here has yielded several key insights into Akira's design, evolution, and operational tactics:

- **Previous Version Weakness & Remediation** – The last Akira release contained a critical vulnerability in its yarrow256 seed generation: it derived entropy solely from QueryPerformanceCounter and a frequency value which is usually constant on modern systems, enabling analysts to bruteforce the seed and decrypt files without paying ransom. The current version replaces this single-source seed with multiple high-precision time values via calls to **xtime_get_ticks** – invoked over 130+ times per encryption cycle (excluding the calls in the **GenerateRandomVal** func), interleaved with context switches, heap reallocations, random value regeneration and multithreaded access – rendering brute-forcing computationally infeasible.
- **Hybrid Encryption Scheme** – Akira's encryptor leverages a dual-stream approach, combining **KCipher2** and **ChaCha8** for data encryption with a **4096-bit RSA** key using **PKCS#1 v1.5** padding for robust key encapsulation.
- **C++ Implementation with Boost.Asio** – Written in modern C++ and built atop **Boost.Asio** for asynchronous, multithreaded I/O, Akira orchestrates concurrent encryption thre hat maximize throughput and significantly elevate the complexity of reverse engineering.

recovery efforts.

Ultimately, this work reaffirms that **proactive reverse engineering** is indispensable in the fight against modern ransomware. Our detailed dissection of Akira transforms an opaque criminal tool into actionable insights, demonstrating our team's analytical rigor and reinforcing our commitment to bolstering cyber defenses against ever-evolving threats.

At this time, we have not identified a vulnerability that permits decryption of the data; however, depending on mistakes made by the threat actor, the data might be recoverable in later versions.

Fully Encrypted Extensions

.4dd,.4dl,.accdb,.accdc,.accde,.accdr,.accdt,.accft,.adb,.ade,.adf,.adp,.arc,.ora,.alf,.ask,.btr,.bdf,.cat,.cdb,.ckp,.cma,.cpd,.dacpac,.dad,.dadiagrams,.daschema,.db,.db-shm,.db-wal,.db3,.dbc,.dbf,.dbs,.dbt,.dbv,.dbx,.dcb,.dct,.dcx,.ddl,.dlis,.dp1,.dqy,.dsk,.dsn,.dtsx,.dxl,.eco,.ecx,.edb,.epim,.exb,.fcd,.fdb,.fic,.fmp,.fmp12,.fmpsl,.fol,.fp3,.fp4,.fp5,.fp7,.fpt,.frm,.gdb,.grdb,.gwi,.hdb,.his,.ib,.idb,.ihx,.itdb,.itw,.jet,.jtx,.kdb,.kexi,.kexic,.kexis,.lgc,.lwx,.maf,.maq,.mar,.mas,.mav,.mdb,.mdf,.mpd,.mrg,.mud,.mwb,.myd,.ndf,.nnt,.nrmlib,.ns2,.ns3,.ns4,.nsf,.nv,.nv2,.nwdb,.nyf,.odb,.oqy,.orx,.owc,.p96,.p97,.pan,.pdb,.pdm,.pnz,.qry,.qvd,.rbf,.rctd,.rod,.rodx,.rpd,.rsd,.sas7bdat,.sbf,.scx,.sdb,.sdc,.sdf,.sis,.spq,.sql,.sqlite,.sqlite3,.sqlitedb,.te,.temx,.tmd,.tps,.trc,.trm,.udb,.udl,.usr,.v12,.vis,.vpd,.vvv,.wdb,.wmdb,.wrk,.xdb,.xld,.xmlff,.abcddb,.abs,.abx,.accdw,.adn,.db2,.fm5,.hjt,.icg,.icr,.kdb,.lut,.maw,.mdn,.mdt

VM extensions

Ignored Extensions

.exe,.dll,.lnk,.sys,.msi,.akira,.arika

Ignored Folders

tmp,winnt,temp,thumb,$Recycle.Bin,$RECYCLE.BIN,System Volume Information, Boot,Windows,Trend Micro,ProgramData

MITRE ATT&CK Techniques used by the ransomware

T1059.001	Command and Scripting Interpreter: PowerShell
T1486	Data Encrypted for Impact
T1490	Inhibit System Recovery
T1559.001	Inter-Process Communication: Component Object Model
T1070.001	Indicator Removal: Clear Windows Event Logs
T1047	Windows Management Instrumenta

- akira_readme.txt
- .arika (ext)
- .akira (ext)

Authors

Hassan Faraz

Specializing in malware reverse engineering and data recovery, Hassan Faraz develops decryption solutions for vulnerable ransomware and creates custom tools to extract data from various file formats such as VHD, VMDK, MDF, backup formats, etc. He has a proven track record of recovering critical file formats with minimal data loss, helping organizations mitigate the impact of cyberattacks.

Andrey Zed

Andrey Zed is a Ransomware R&D Engineer with a background in Ruby development,

Laura Pompeu

Laura Pompeu is an editor and content strategy leader at Porthas, bringing over 10 years of digital media experience. Leveraging her background in journalism, SEO, and marketing, Laura shapes cybersecurity and technology content to be insightful yet accessible.

Leave a comment

Your email address will not be published. Required fields are marked *

PORTHAS INC

Home Services ▾ About Us ▾ Blog Contact us

Questions? Talk to Us
+1 (844) 969-6683

GET HELP

Email

Save my name, email, and website in this browser for the next time I comment.

POST COMMENT

Search ...

Categories

CASE STUDY (3)

CYBER ATTACK (4)

DATA BREACH (5)

DFIR REPORT (2)

INCIDENT RESPONSE (4)

NEWS (3)

RANSOMWARE (11)

UNCATEGORIZED (1)

PORTHAS INC

Home Services ▾ About Us ▾ Blog Contact us

Questions? Talk to Us

+1 (844) 969-6683

GET HELP

16 Billion Password Leak: How to Protect Your Business

Akira Ransomware: In-Depth Technical Analysis

What is Cyber Incident Response & Immediate Steps