https://web.archive.org/web/20251013150404if_/https://www.porthas.com/blog/b0-ransomware-decryption/

Investigation & Decryption – Porthas Inc

By Mohamed Talaat

Published: 2025-04-17 · Archived: 2026-11-25 02:04:10 UTC

Breaking the unbreakable: Story of B0 software group ransomware

This is how deep technical curiosity, DFIR expertise, and persistence led to the downfall of B0 Software Group's ransomware. In this article, we walk through the full investigation from start to finish: from the initial response and behavioral analysis to reverse engineering and decryptor development. With annotated code snippets, screenshots, and step-by-step breakdowns, we reveal not just what was done, but why it worked.

If you're looking for a real-world look at how threat intelligence and malware analysis can turn the tide in ransomware recovery, this is it.

Known Tactics, Techniques, and Procedures

Since ransomware groups can often sell their malware to multiple affiliates. These affiliates can have different flavors and attack methodologies for how they breach a targeted network. As such, there is no single set of TTPs that can generally cover how B0 software group ransomware can end up on your network. In this case, however, the DFIR investigation highly suggests that brute-forcing an internet-facing RDP (RDP brute-forcing) was the initial entry point.

The assessment was based on the following factors:

- Evidence of a brute force attack. Found repeated failed logins to the Administrator account on the compromised server. The brute force attempts originated from external IP addresses.
- The default RDP port 3389 was directly exposed to the internet at the time of the incident. The default RDP port is frequently an attractive attack vector and can be easily discovered by automated port scanners such as Nmap.

DFIR Investigation Conclusion

As for activities within the network and lateral movement, the DFIR investigation concluded the following points:

- Network discovery/reconnaissance. Upon gaining access to the server, the threat actor used the network reconnaissance tools Advance IP Scanner and kportscan 3.0 to identify additional devices to which to move laterally. However, because the server was the only device within a segmented, isolated environment, further lateral movement was unsuccessful.
- Account usage and privilege escalation. The threat actor gained access to the privileged Administrator account on the server via a brute force attack. Upon successfully logging into the system, they also created the account B0Us. The threat actor also executed the credential harvesting tool, Mimikatz, on the server.

This is a credential dumping tool, typically used by threat actors to obtain any available credentials stored within the memory of a target system.

- Ransomware deployment. The threat actor transferred the compressed file ‘pth.rar’ to the system, which contained their toolkit, including the ransomware executable ‘b0.exe’. The compressed file was initially placed in the directory ‘C:\users\b0us\desktop\’ and then subsequently extracted using the file compression tool, WinRAR. The threat actor placed their ransomware binary within the folder ‘C:\users\public’, manually executing this on the server leading to the encryption of files on the system and creation of ransom notes.

Exfiltration Analysis

We assess that the threat actor **did not exfiltrate data** from the impacted server. This assessment is based on the following factors:

- No evidence of file transfer tooling. S-RM found no evidence of data transfer tooling that could have been used by the threat actor to exfiltrate data from the server.
- No evidence of data access or staging activities. S-RM reviewed data extracted from the system and did not identify any conclusive evidence typically preceding exfiltration, such as data reconnaissance and compression. While we found evidence of interaction with the server’s data drives, it is likely that this occurred in the context of pre-encryption activities.
- Threat actor claims. The threat actor did not claim to have exfiltrated any data from the server in their ransom note. Further, S-RM found no evidence of a leak site maintained by the threat actor to publish victim data as part of data extortion tactics.

Limitations

- Due to the hosted nature of the server, S-RM could not obtain access to network or firewall logs, typically providing insight into data traffic and volumes. We were therefore unable to determine whether there any significant spikes in outbound data during the timeframe of the incident consistent with exfiltration activities.
- The evidence indicates that a limited number of files were accessed by the threat actor, however there is no accompanying evidence that would show that this data was transferred out of the system.

Technical analysis

The ransomware was written in Golang, making it more challenging to reverse engineer compared to other ransomware variants which are in C/C++.

The compilation timestamp of the analyzed sample was set to zero, which might have been a tactic used by the authors to confuse analysts and make it more difficult to determine when the sample was compiled. Notably, the **ransomware includes its own kill switch**. It calculates the time difference between a hardcoded UTC timestamp of **March 24, 2025, at 00:00:00**, and the time when the sample is executed. The ransomware checks if this time difference falls within a 3-day range. If it does, the ransomware continues its normal execution. If not, it immediately restarts the system by executing the command **“shutdown /r /t 0”**.


Screenshot of a code snippet written in C-style syntax, possibly decompiled from Go, that checks whether the current date is within three days of March 24, 2025. If the time difference is within the range, it calls a function main_HavePassword. If not, it prints a message to standard output and initiates a system shutdown using main_restartSystem and a sleep delay. The code appears to perform a forced system restart if the time condition isn't met.

After confirming that the process runs within the allowed time range, it moves on to another check. During this check, it prompts the user for a password and a network ID. The provided password is then hashed using SHA-256 and compared against a hardcoded hash. If the hashes match, the execution will proceed; otherwise, it will prompt for the password again, up to three times. If all three attempts fail, the process will exit.


Screenshot of a decompiled code segment that verifies a hashed password. The function main_hashPasswordLogin generates a hash, and if the hash length equals 0x40, it is compared using runtime_memequal against a hardcoded hash value. If the hashes match, the function main_HaveRun is executed, indicating successful authentication. If not, an error message is printed to standard output using fmt_Fprintln. The code includes a hardcoded SHA-256-like hash and appears to validate login credentials securely before allowing further execution.

Upon successfully completing all the previously mentioned checks, the process moves on to configure the setup. First, it initializes a pseudo-random number generator using the current system time obtained by calling “time.Now.UnixNano().” Next, it generates a random nine-character string that will later be used for the key derivation function. This key will be crucial for the decryption and recovery of ransomware, which we will discuss later.


Screenshot of decompiled source code in a C-like language that seeds a pseudo-random number generator using the current system time. The function time_Now() retrieves the current time, which is then manipulated through bitwise operations to handle negative values and mask it to 30 bits. The code calls math_rand_ptr_Rand_Seed to initialize the random number generator with a seed derived from the processed time value. Finally, it generates a 9-character random string using BOClient_GetRandomString_RandomString. Comments in green explain the steps, such as time retrieval, seed generation, and random key creation.

Next, it will verify the existence of the JSON file “telemetry.ASM-WindowsDefault.json” located in “C:\Windows\System32”. This JSON file contains the ransomware configuration, including the initial key (a random nine-character string), the extension, and the Network ID.


JSON config file containing the extension, key and the network ID
JSON config file containing the extension, key and the network ID.

If the file is found, it continues by reading the file and setting the corresponding global variables for ransomware configuration.


Screenshot of C/C++ source code for ransomware configuration logic. The code checks for the existence of a key file using os_Stat, reads the key file and sets global variables for ransomware configuration if it exists, and creates a new key file with configuration globals if it does not. Functions shown include BQClient_GetConfig_ReadKey and BQClient_GetConfig_CreateKeyFile. Variable names like v9, v10, v11, v12, v13, v14, v18, and qword_11D8198 are visible, along with comments describing the process.
Configuration check and setup.

Similar to the key file, it will check for the existence of a file “Icon.png”, located in the Windows directory “C:\Windows”. Same as before, if the file is not found, it proceeds to create it. The icon file contents are stored as base64 encoded text that will be decoded and written to the file at run-time.

If the file was not found, it then creates the file “Icon.png” and sets the default icon for files with the ransomware’s extension (as specified in the ransomware configuration) to this “Icon.png.”


Screenshot of disassembled ransomware code logic. The code checks for the existence of "C:\Windows\Icon.png" using os_Stat, and if not found, creates the icon file, retrieves configuration for file extensions, and sets a default icon for files with the ransomware's extension. If the icon file exists, the code proceeds to stop important services and kill processes, including IIS, SQL, and other critical services. Function names like BQClient_CreateIcon_CreateIcon, BQClient_StopImportantServices_StopImportantService, and BQClient_KillProcess_KillProcess are visible, along with descriptive comments.
Check for “Icon.png”, and create if not found.

Process and Service targeting

Before proceeding with key derivation and encryption, the ransomware will attempt to stop related services for Windows IIS, MSSQL, and MySQL by executing the commands listed below.

BlockListed services

- cmd /C iisreset stop
- cmd /C NET STOP IISADMIN
- cmd /C net stop WAS
- cmd /C NET stop MSSQLSERVER
- cmd /C NET stop \”SQL Server (MSSQLSERVER)\”
- cmd /C net stop MSSQL$SQLEXPRESS
- cmd /C net stop SQLSERVERAGENT
- cmd /C net stop mysql
- sc stop W3SVC
- sc stop WAS
- sc stop IISADMIN

It’s important to note that not only it targets specific services by name but also attempts to obtain a list of running services and processes using the commands “sc query” and “tasklist.” Additionally, it tries to terminate any service

or process that contains the word “SQL” in its name.

Anti-forensics measures

The ransomware deploys a variety of anti-forensic measures, to avoid leaving behind any leads that can be used in a forensic investigation. The following is a list of the anti-forensics’ measures taken by the ransomware to wipe any evidence.

1. Delete the Windows IIS logs located in the directory: **C:\inetpub\logs**.
2. Empty the Recycle Bin.
3. Constantly clears Windows Event Logs.
4. Constantly Clear the contents of the Windows Temp folder by running the following command:

```
Del /S /F /Q %Windir%\Temp
```

5. Create a registry entry under the Run key to clear Windows Event Logs upon user login. Use the following command:

```
powershell.exe -Command New-ItemProperty -Path HKCU:\Software\Microsoft\Windows\CurrentVersion\Run -Name De
```

This will ensure that the Windows Event Logs are cleared each time the user logs in.

Furthermore, the following commands are written to a file called “Del.cmd,” which is saved in the system directory C:\Windows\System32\ and executed. This script will delete the ransomware executable as well as the Windows Event logs.

Contents of Del.cmd, that will be executed after encryption to wipe any traces of its activity

```
 timeout /t 10
del /f ransomware_executable_path
powershell "wevtutil el | Foreach-Object {wevtutil cl "$_"}
```

It is noteworthy to mention that the ransomware does not delete the Windows Event Logs; instead, it continuously clears them during execution to erase any traces of its activity. The same behavior applies to the Windows Temp folder.


Ransomware execution flow in disassembled code, showing calls to main_FindFile, BQClient_RunCmdExecutable_DelTemp (wipes Windows Temp folder), BQClient_RunCmdExecutable_ClearLog (clears event logs), and BQClient_RunCmdExecutable_DeleteEndedFiles (executes Del.cmd to clear logs and delete the ransomware executable after encryption). Code snippet illustrates post-encryption cleanup routines.

Key Derivation

The initially generated random nine-character string is used to generate the AES-GCM key, which is then used during encryption. The ransomware first uses the hashids library to generate a hashed ID using the random string as the salt and 1,000,000,000(0x3B9ACA00) . This library can generate hashed/obfuscated IDs from numbers. The hashed ID's size is 7 bytes, and its character set is limited to 'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ1234567890'.


Code snippet demonstrating salt generation for ransomware. The algorithm initializes a salt using alphanumeric characters and a key, and then encodes an ID using the 'github.com/speps/go-hashids' library. The initialization of arrays hd(0), hd(1), hd(4), hd(3), and hd(2) is shown. Function call to 'github_com_speps_go_hashids_NewWithData' indicates a custom hashing implementation for obfuscation or unique ID creation within the ransomware.

The hashed ID is then hashed in the following order: SHA1 → MD5 → SHA512 → SHA1 → MD5 → MD5 → SHA1 to generate a password that will be used in the password based key derivation function. The hashing functions also encode the hashed data before returning it.


GoLang source code snippet showing a hashing function within a ransomware program. The function B0Client_Hashing_GetCode takes a hashed ID as input and generates SHA1, MD5, and SHA512 hashes using custom functions like B0Client_Hashing_Sha1Generator and B0Client_Hashing_GetMD5Hash. The function performs multiple rounds of hashing to strengthen the encryption process.

After this, PBKDF2 is used to derive the AES-GCM key from the password. The salt is the same as the password; 100,000 iterations are used, and the key length is 32 bytes. SHA512 is used as the hash function for the password based key derivation function. Notably, if a null value is passed as the salt(that indicates for some reason it failed to generate and hash the random nine-character string), the ransomware generates a random salt using the crypto/rand package; however, if this occurs, it is impossible to decrypt the file even with the threat actor's decryptor, since the ransomware does not store the salt somewhere. Regardless, this scenario would not happen, as the third argument is always the password converted to bytes in the current version of the ransomware.


GoLang code snippet showing the generation of a salt for encryption within a ransomware program. The code initializes 'salt2' with a length of 32 bytes using 'runtime.makeslice' and populates it with random data using 'crypto_rand.Read'. It then converts both the password and salt to byte slices for use in the 'golang_org_x_crypto_pbkdf2.Key' function, which generates a derived key using the PBKDF2 algorithm with SHA512. Constants such as 100000LL (iterations) and 32uLL (key length) are visible.

The resulting key is used to encrypt the files with AES-GCM, and the same key is used for each file.

Ransom Note

As is common to ransomware groups, a note with instructions on how to contact the actor is dropped in each driver's root directory. The victim can negotiate payment options via a communication channel of the actor's choosing. The note is dropped in two forms : a text file, How_To_Recovery_Files.txt, and an HTML file also named How_To_Recovery_Files.html with slightly different content.

It is worth mentioning that the Network ID(serves a unique identifier to the victim) provided earlier in the early of its execution, is added to the contents of the ransom note before it's written to disk.


Disassembled code snippet showing how the ransomware drops ransom notes in the root directory of each drive letter, with a highlighted comment indicating this behavior and supporting function calls for deleting temporary files and clearing logs.
How the ransom notes are dropped in the root directory of each drive letter.


Screenshot of the HTML ransom note used by the B0 Software Group ransomware. The note informs the victim that files have been encrypted with a unique identifier and promises data restoration upon payment. It includes sections titled 'What happened,' 'Risks caused by non-cooperation' listing 18 potential consequences of data loss, 'What guarantees you have,' and a warning labeled 'Attention!' advising against third-party recovery attempts.
HTML version of the ransomware note.


Text-based ransom note from the B0 Software Group, instructing victims to install the Tox messaging app and contact the attackers via a Tox ID. The note explains Tox as a decentralized, encrypted messaging platform and warns of potential communication issues. It provides a .onion Tor website address for further instructions and emphasizes downloading the Tor browser from the official site. It also mentions that the system password may have been changed.
Text version of the ransomware note.

File Encryption

The *main_FindFile* function retrieves a list of the root directory paths for each available drive letter, and then calls *main_run* on each.


C/C++ code snippet showing a loop that iterates through a list of drives obtained from the function BOClient_GetAllDrivesList_GetDrives(). The code initializes variables, then uses a while loop to call a function named main_run with parameters including drive path, key sizes, and other variables. The loop updates pointers and indices to process each drive, and finally returns the drive pointer. Some function parameters (a5, v6, v7) are highlighted in orange.

The *main_run* uses *filepath.Walk()* to recursively walk through all the files, and directories. The callback function passed as an argument to *filepath.Walk()* performs some checks on the file path to decide whether the file can be encrypted or not. It first checks whether any of the following blacklisted paths are in the filepath, and if they are, the callback function returns without encrypting the file.


A screenshot of a C/C++ code segment that checks if a result variable is false, then assigns various Windows system and program directory paths to variables using the function runtime_concatstring2. The paths include directories such as Windows, ProgramData, Program Files, Program Files (x86), Internet Explorer, WindowsApps, Embedded Lockdown Manager, Microsoft, and Recycle Bin. Directory names and comments are highlighted in green.

Below we summarize the list of paths excluded by the ransomware during encryption.

BlackListed Filepaths:

- \WindowsPowerShell\Modules\
- \Windows
- \ProgramData
- \Program Files\Reference Assemblies\
- \Program Files\Common Files\
- \Program Files\Internet Explorer
- \Program Files (x86)\Reference Assemblies\
- \Program Files (x86)\Common Files\
- \Program Files (x86)\Internet Explorer\
- \Program Files\WindowsApps\
- \Program Files\Embedded Lockdown Manager\
- \Program Files\VMware\
- \ProgramData\Microsoft\
- \$Recycle.Bin\

The next check determines whether the ransomware extension is at the end of the file, and whether the extension is *.ini* or *.sys.* If it is either of these, the file is skipped. After this, it checks whether the extension is *.bak* or *.log*, and if it is, it proceeds with deleting the file and returning. If a file doesn't pass any of these checks, the encryption function is called on it. Since these checks do not check whether the file is the ransomware executable itself, the ransomware ends up encrypting its own executable. However, it fails to remove the plaintext executable after the encryption process using *os.remove*.


A screenshot of a C/C++ code block that processes files based on their extensions. The code skips files with .ini and .sys extensions, removes files with .bak or .log in their names, and otherwise encrypts files using the BOClient_Encryptor_Encrypt function. Comments in green explain the logic, while some function parameters and variables are highlighted in orange and blue for emphasis.

The file encryption function first prints the filepath. After which, it attempts to open the file. If that fails, the ransomware just skips the file instead of trying to kill the process that's using the file. The ransomware also checks that the file is not a directory after opening it.


A screenshot of C/C++ code that checks if a file is encrypted using the BOClient_Encryptor_IsFile function. If the file is encrypted or inaccessible, it prints an error message using fmt_Error indicating the file is locked or in use. The code opens the file, checks for errors, and formats an error message if needed. Function parameters and variables are highlighted in different colors to differentiate them.

The ransomware initiates the encryption process by generating an AES cipher object and initializing it with the previously generated key. It uses GCM mode with a 16-byte tag size and a 12-byte nonce size. The nonce is randomly generated using the *crypto/rand* package.


A screenshot of C/C++ code that uses crypto_aes_NewCipher to create a new cipher. If successful, it prints an error message using fmt_Error; otherwise, it uses crypto_cipher_newGCMWithNonceAndTagSize and crypto_rand_Read to generate a nonce and tag. A memory address MEMORY[0x14] is highlighted in red.

After this, the ransomware creates a new file named by appending *.id-*. to the original filename. The initially generated key is stored both in the configuration file and within each encrypted file's name. The ransomware then writes a 16-byte marker, consisting of null bytes, followed by the random nonce, to the new file. After this, it reads data from the original file in chunks of 0x400000 bytes, encrypts each chunk using the *Seal()* function, which handles both encryption and authentication. The resulting ciphertext, along with the 16-byte authentication tag, are both written to the new file. Once all chunks have been encrypted and written, the original file is deleted using *os.remove*.


A screenshot of disassembled x86-64 code showing a conditional jump based on a comparison. If the value in the 'rax' register is equal to 400000h, the code jumps to the address 'loc_82770F'. Below, a series of 'mov' instructions manipulate data on the stack related to error handling and file writing, calling functions like 'os_ptr_File_Write' and 'fmt_Error'. Code blocks are connected by arrows to show the flow of execution.

Encrypted file structure

The structure of the encrypted file is quite straightforward. At the beginning of the encryption process, a 16-byte marker (highlighted in purple) consisting of NULL bytes is added to the start of the file to indicate that it is encrypted. Following this marker, there is a 12-byte nonce (highlighted in Red in the screenshot below) value that remains consistent throughout the entire file. This nonce is stored within the file for use during decryption later. The rest of the data is just the file's encrypted data (highlighted in light green in the screenshot below).


Encrypted file structure highlighting marker, nonce, and encrypted bytes.
Encrypted file structure highlighting marker, nonce, and encrypted bytes.

Weakness

There are two major weaknesses in the encryption. The first, and most obvious, is that the AES-GCM key can be easily derived from the initial key (randomly generated nine-character string), which is unique to each victim and generated at run-time, then saved to disk later.

The second weakness is less apparent. Although the ransomware uses AES-GCM to encrypt blocks (which combines AES-CTR for encryption and GMAC for authentication), it doesn't generate a new nonce for each block. As a result, the same keystream is reused for each block within a file. If we know the plaintext for one block, we can recover the keystream by xoring it with the corresponding ciphertext and then use that keystream to decrypt all subsequent blocks. However, exploiting this vulnerability is not straightforward, since it requires a significant amount of known plaintext (0x400000) bytes.

Decryptor

We have created a b0 ransomware decryptor for those affected by this ransomware. Errors are logged in .\log.txt. Below is the usage guide.

Usage of decryptor:

- dirpath string: Path to the directory to recursively decrypt files
- key string: Key to use for decryption, in the following filename test.pdf.id-QVKGBICKS.B0-3e72d, QVKGBICKS is the key.
- path string: Path to the encrypted file

Indicators Of Compromise

Known hashes

cea74b854b6cd161884e5ced5c8b7ba44c1064ff22703667cc913d9a67f1767b (SHA-256)

File Paths

Ransomware executable	B0.exe
Ransomware configuration written to disk at run-time	C:\Windows\System32\ telemetry.ASM-WindowsDefault.json
Ransomware icon	C:\Windows\Icon.png
Batch script	C:\Windows\System32\Del.cmd
Ransom note in text and HTML versions	C:\How_To_Recovery_Files.txt C:\How_To_Recovery_Files.html
Registry entry created to clear Windows Event logs on user login	HKCU:\Software\Microsoft\Windows\CurrentVersion\Run\ DeleteLogWithStartUp

Conclusion

This investigation is a powerful reminder that even in an ecosystem dominated by rapidly evolving ransomware threats, there’s still room for skilled analysis, persistence, and creative problem-solving to make a real impact. While not every case ends in a decryptor, this one proves that deep malware analysis and DFIR expertise can tip the scales.

- 

Mohamed Talaat is a self-taught security researcher specializing in malware analysis, threat intelligence, and tracking APT groups and crimeware. With a background in engineering and over three years of independent research, his focus is on ransomware recovery, evaluating recovery paths, analyzing encryption schemes, and occasionally developing decryptors. Passionate about reverse engineering and low-level analysis, he also writes YARA rules to help detect and hunt emerging threats and campaigns.

- 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.

Source: https://web.archive.org/web/20251013150404if_/https://www.porthas.com/blog/b0-ransomware-decryption/