

```
{
  "payload": {
    "allShortcutsEnabled": false,
    "fileTree": {
      "WikiLoader": {
        "items": [
          {
            "name": "Images",
            "path": "WikiLoader/Images",
            "contentType": "directory"
          },
          {
            "name": "WikiLoader Shellcode pt2.md",
            "path": "WikiLoader/WikiLoader Shellcode pt2.md",
            "contentType": "file"
          },
          {
            "name": "WikiLoader Shellcode pt3.md",
            "path": "WikiLoader/WikiLoader Shellcode pt3.md",
            "contentType": "file"
          },
          {
            "name": "WikiLoader notepad.md",
            "path": "WikiLoader/WikiLoader notepad.md",
            "contentType": "file"
          }
        ],
        "totalCount": 4
      },
      "items": [
        {
            "name": ".obsidian",
            "path": ".obsidian",
            "contentType": "directory"
          },
          {
            "name": "Pikabot",
            "path": "Pikabot",
            "contentType": "directory"
          },
          {
            "name": "WSHRAT",
            "path": "WSHRAT",
            "contentType": "directory"
          },
          {
            "name": "WikiLoader",
            "path": "WikiLoader",
            "contentType": "directory"
          },
          {
            "name": "README.md",
            "path": "README.md",
            "contentType": "file"
          }
        ],
        "totalCount": 5
      },
      "fileTreeProcessingTime": 7.176824,
      "foldersToFetch": [],
      "repo": {
        "id": "675143377",
        "defaultBranch": "main",
        "name": "MalwareAnalysisReports",
        "ownerLogin": "VenzoV",
        "currentUserCanPush": false,
        "isFork": false,
        "isEr08-05T23:42:05.000Z",
        "ownerAvatar": "https://avatars.githubusercontent.com/u/107503502?v=4",
        "public": true,
        "private": false,
        "isOrgOwned": false,
        "symbolsExpanded": false,
        "treeExpanded": true,
        "refInfo": {
          "name": "main",
          "listCacheKey": "v0:1691279081.0",
          "canEdit": false,
          "refType": "branch",
          "currentOid": "e5a302131a634bb67d77dc8e5068d0e3c14018notepad.md",
          "currentUser": null,
          "blob": {
            "rawLines": null,
            "stylingDirectives": null,
            "csv": null,
            "csvError": null,
            "dependabotInfo": {
              "showConfigurationBanner": false,
              "configFilePath": null,
              "networkDependabotPath": "/VenzoV/MalwareAnalysisReports/network/updates",
              "dismissConfigNotice/dependabot_configuration_notice",
              "configurationNoticeDismissed": null,
              "displayName": "WikiLoader notepad.md",
              "displayUrl": "https://github.com/VenzoV/MalwareAnalysisReports/blob/main/WikiLoader/WikiLoader%20notepad.md?raw=true",
              "headerInfo": {
                "blobSize": "14.5 KB",
                "deleteTooltip": "You must be signed in to make or propose changes",
                "editTooltip": "You must be signed in to make or propose changes",
                "deleteInfo": {
                  "deleteTooltip": "You must be signed in to make or propose changes",
                  "editInfo": {
                    "editTooltip": "You must be signed in to make or propose changes"
                  }
                }
              },
              "ghDesktopPath": "https://desktop.github.com",
              "isGitLfs": false,
              "gitLfsPath": null,
              "onBranch": true,
              "shortPath": "e3d1394",
              "siteNavLoginPath": "https://github.com/VenzoV/MalwareAnalysisReports/blob/main/WikiLoader/WikiLoader%2520notepad.md",
              "level": 1,
              "text": "Sample Information",
              "anchor": "sample-information",
              "htmlText": "Sample Information"
            },
            {"level": 2, "text": "ZIP file contents and \nnotepad", "anchor": "zip-file-contents-and-notepad", "htmlText": "ZIP file contents and \nnotepad"},
            {"level": 2, "text": "Static Analysis", "anchor": "static-analysis", "htmlText": "Static Analysis"},
            {"level": 2, "text": "Information Gathering", "anchor": "information-gathering", "htmlText": "Information Gathering"},
            {"level": 1, "text": "Dynamic Analysis", "anchor": "dynamic-analysis", "htmlText": "Dynamic Analysis"},
            {"level": 3, "text": "Getting Kernel32.dll & GetProcAddress", "anchor": "getting-kernel32dll-getprocaddress", "htmlText": "Getting Kernel32.dll & GetProcAddress"},
            {"level": 3, "text": "String builder and Control flow manipulation", "anchor": "string-builder-and-control-flow-manipulation", "htmlText": "String builder and Control flow manipulation"},
            {"level": 3, "text": "Thread manipulation", "anchor": "thread-manipulation", "htmlText": "Thread manipulation"},
            {"level": 3, "text": "Decrypting Shellcode", "anchor": "decrypting-shellcode", "htmlText": "Decrypting Shellcode"},
            {"level": 2, "text": "References", "anchor": "references", "htmlText": "References"}
          ],
          "lineInfo": {
            "truncatedLoc": "411",
            "truncatedSloc": "278",
            "mode": "file",
            "image": false,
            "isCodeownersFile": null,
            "isPlain": false,
            "isValidLegacyIssueTemplate": false
          },
          "issue-and-pull-request-templates",
          "issueTemplate": null,
          "discussionTemplate": null,
          "language": "Markdown",
          "languageID": 222,
          "large": false,
          "loggedIn": false,
          "planSupportInfo": {
            "reposFork": null,
            "repoOwnedByCurrentUser": null,
            "requestFullPath": "/VenzoV/MalwareAnalysisReports/blob/main/WikiLoader/WikiLoader%20notepad.md",
            "dismissActionNoticePath": "/settings/dismiss-notice/publish_action_from_dockerfile",
            "releasePath": "/VenzoV/MalwareAnalysisReports/releases/new?marketplace=true",
            "showPublishActionBanner": false
          },
          "rawBlobUri": "https://github.com/VenzoV/MalwareAnalysisReports/raw/main/WikiLoader/Wiki"
        }
      }
    }
  }
}
```

Sample Information

I stumbled upon this sample checking the following post:



The following hash is for the malicious .dll "MimeTools.dll"

\\.\.\.\.\.\.\.\.\.\.\.\.\.\.\.\.

SHA256

67283e154b86612e325030e5a5f7995a6fe552d20655283ea5de8b53ff405f69

Following the hashes for the .zip file which contains the .dll.

\\.\.\.\.\.\.\.\.\.\.\.\.\.\.\.\.

SHA256

bef04e3b2b81f2dee39c42ab9be781f3db0059ec722ae3b5434c2e63512a68

\n

ZIP file contents and \"notepad\"

\n

The zip file initially contains files that mimic the notepad++ file structure and files. The end game is for the \"notepad.exe\" to side load the malicious .dll \"mimetools.dll\"


\n

Malicious zip file contents:

\n

 \"image\"

\n

Legitimate notepad++ .exe folder structure.  \"image\"


\n

The malicious .dll is located also in the same path notepad++ contains the file. This is of course so that the sideloading can work. The path for the .dll:

\n

```
\n  plugins\\mimeTools\\  
\n
```

\n

Comparing hashes on VT or any other site like unpac.me, we can observe that they are different and one of them is also detected by many engines.  \"image\"

\n

Static Analysis

\n

Using IDA or any decompiler won't really help initially. The sample uses a lot of control flow obfuscation essentially changing calls for jmp instructions. This will confuse IDA and we don't get clear disassembly view nor decompiler. So, for the moment we will go with dynamic analysis with x64 dbg.

\n

Information Gathering

\n

Some information based on OSINT and other resources just to get ahead and have some hits on what to look for. I was able to find only one report from proofpoint with some useful information. Also online sandboxes gives us some hints of behaviors to expect.

\n

We will expect PE parsing to occur:

\n

 \"Image\"

\n

I also observed from IDA some potential stack strings, I ran FLOSS to try and extract some. Following the results, which we will use to help our analysis. Although it may not be useful in the end.

\n

```
FLOSS Stack Strings\n\nFLOSS decoded 1 strings\n !\"#$%&'()*+,-./0123456789:;<=>?  
@ABCDEFGHIJKLMN0PQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~\n\nFLOSS extracted 67  
stackstrings\n\nXm21X0k}Z\n\nVirtualAlloc\n\nCryptStringToBinary\n\nCryptStringToBinary\n\nXm28mBHjNU7\n\nXm28mBHjNU7\n\nCryptStri  
ngToBinary\n\nXm28mBHjNU7\n\nHjGwPX65nXm21X0kX4nMqex28mBHjNU7\n\nXm28mBHjNU7\n\nLoadLibrary\n\nZBAA\n\nVirtualVirtualAlloc\n\nXm21X0k}Z\n\nVirtualA  
lloc\n\nVirtualAlloc\n\nXm28mBHjNU7\n\nVirtualFree\n\nVirtualFree\n\nXm28mBHjNU7\n\nHjGwPX65nXm21X0kX4nMq  
x28mBHjNU7\n\nLoadLibrary\n\nXm21X0k}Z\n\nVirtualFree\n\nHjGwPX65nXm21X0kX4nMqex28mBHjNU7\n\nLoadLibrary  
A\n\nCryptStringToBinary\n\nLoadLibrary\n\nXm28mBHjNU7\n\nVirtual\n\nLoadLibrary\n\nVirtualFree\n\nHjGwPX65nXm21X0kX4nMqex28mBHjNU7\n\nXm21X0k}Z\n\nv  
irtualFree\n\nVirtual\n\nXm28mBHjNU7\n\nVirtual\n\nXm28mBHjNU7\n\nCryptStringToBinary\n\nLoadLibrary\n\nLoadLibrary\n\nXm28mBHjNU7\n\nLoadLibrary\n\nHjGwPX6  
5nXm21X0kX4nMqex28mBHjNU7\n\nXm28mBHjNU7\n\nVirtualAlloc\n\nXm28mBHjNU7\n\nCryptStringToBinary\n\nXm28mBHjNU7\n\nXm21X0k}Z\n\nHjGwPX65nXm21X0kX4nM  
qex28mBHjNU7\n\nXm21X0k}Z\n\nCryptStringToBinary\n\nVirtualAlloc\n\nVirtual\n\nVirtualFree\n\nCreateThread\n\nVirtualFree\n\nLoadLibrary\n\nVirtual  
Free\n\n
```

\n

Dynamic Analysis

\n

Getting Kernel32.dll & GetProcAddress

\n

First thing the malware gets a handle to the PEB from the TIB, and then sets up relevant offsets to access the LIST_ENTRY InMemoryOrderModuleList structure. As per microsoft docs it is a double linked list that contains the loaded modules for the process. This will be used by the malware to go through the dlls loaded. ntdll.dll is the first .dll of the process, followed by kernel32.dll.

\n

- \n
 - Offset 0x60 is passed to RAX register
 - \n
 - RAX is used with gs: special register to obtain the PEB location.
 - \n
 - Offset 0x18 is passed to RBX
 - \n
 - Offset RBX+RAX is used to access PEB_LDR_DATA struct and saved to RBX
 - \n
 - Offset 0x20 is passed to RAX
 - \n
 - Offset RBX+RAX now leads to LIST_ENTRY InMemoryOrderModuleList
 - \n
 - The first of the list InMemoryOrderModuleList is now saved in r12 register, this will be used to check for end of list. Since it is a doubled link list, if the code goes through all the modules, it will eventually end up at the start.
 - \n
 - Lastly it gets the 0x40 offset to add to the InMemoryOrderModuleList. This is for the Dllbase address.
 - \n

\n

 "image"

\n

The objective of this first loop is to find kernel32.dll. By walking the PEB, we expect to find the following module order:

\n

- \n
 - Executing Process
 - \n
 - ntdll.dll
 - \n
 - kernel32.dll
 - \n

\n

So, the code will have to go through 3 forward pointers. The way the malware checks if it has the right dll reference is through series of jumps. A certain offset is added to the Dllbase address, this will be a letter in the name, then it is compared to a letter. The malware will check these letters for the kernel32.dll

\n

- \n
 - 4b->'K'

- \n
- 45->'E'
- \n
- 4c->'L'
- \n
- 32->'2'
- \n
- 4c->'L'
- \n
- 4c->'L'
- \n

\n


\n
 Now that the malware has the base address to Kernel32.dll, it will go through all the functions to search for GetProcAddress.\nIt parses the PE header of kernel32.dll to reach the export table.

- \n
1. Offset 3C to reach PE header pointer with value F8.
 - \n
 2. Offset 88 to reach the export table, this value is calculated with: RVA of export table - F8. In our case 0x180-F8=0x88
 - \n
 3. Offset 0x20 to finally reach the exported functions names table.
 - \n

\n


\n


\n
 To check it performs some calculations to build a hex value which correspond to the first 8 bytes of \"GetProcAddress\" keeping in mind endianness.\nIt then loops through all the functions in kernel32.dll to perform compare.

- \n
1. Calculates the first 8 bytes in hex for GetProcAddress string (GetProcAddress)
 - \n
 2. Loops through all the functions and compares the 8 bytes to r9
 - \n

\n


\n


\n
 Next, the malware need to retrieve the function address. To to this it will make use of the ordinal name stored in RCX. The ordinal will be searched withing the AddressOfFunctions struct to get the value.

\n


\n


\n
 So all this PE parsing is to retrieve finally GetProcAddress and use this API to load others.

\n

String builder and Control flow manipulation

\n

Next the malware builds the string \"VirtualAlloc\" and proceeds to user GetProcAddress to fetch the function from kernel32.dll.\nThis is done through jmp instructions, where the function needed is loaded into the appropriate registry.

\n

The string is built by using the lower registry al which contains 1 byte.\nIt simply adds two hex values to generate a char. At the end the null byte is used as terminator.

\n

 \"Image\"

\n

The block of code after this is responsible for the following:

\n

\n

- Sets up the arguments for GetProcAddress in RCX and RDX in this case the module kernel32.dll and API to fetch VirtualAlloc

\n

- Return address following the jmp is saved to rax which is pushed to the stack, this makes it so that when the jmp returns the return address will be on the stack. This points simply to the code following the jmp rax instruction. This is repeated throughout the next jmp instructions to other API.

\n

- Finally it jmp to the GetProcAddress API with the args mentioned above returning the address to VirutalAlloc from kernel32.dll

\n

\n

 \"Image\"

\n

 \"Image\"

\n

Next thing of course, the VirtualAlloc is called to allocate some memory space with PAGE_EXECUTE_READWRITE.

\n

 \"Image\"

\n

Allocated memory buffer:

\n

 \"Image\"

\n

Through the entire sample, all API strings to be fetched or DLLs to be loaded are built in a similar fashion as shown above.\nAll calls to any API are changed with JMP instructions where the return value is pushed onto the stack before execution, this is so following the ret from the JMP, the code can continue.

\n

Thread manipulation

\n

Using the same string creation as for VirtualAlloc() and same usage of GetProcAddress(), the malware proceeds to call GetCurrentThreadID() API.

\n

 \"Image\"

\n

Following OpenThread() call.

\n

 \"Image\"

\n

Finally a new thread is created by calling CreateThread(). The arguments passed:

\n

\n

- Pointer to start address in R8

\n

- Pointer to parameter passed to thread in R9, in this case seems to be the ID handle to that thread.

\n

\n

 "Image"

\n

 "Image"

\n

The malware will then follow by suspending the main thread and leave execution to the newly created one. For debugging purposes, the debugger was acting unpredictably maybe because of race conditions and threads colliding. Once the code reached the new thread start, I manually killed the main thread this seems to have worked but not 100% sure if it is just a fluke.

\n

Decrypting Shellcode

\n

Shellcode is decrypted from the certificate.pem file. Malware uses CreateFileA() to open the file and then ReadFile() to read the contents. It is then decrypted using the key, with AES in CBC mode.

\n

\n

key: HJGwPX65nXm21XOkX4nMqex28mBHjNU7

\n

\n

The decryption functions are not custom, and uses the bcrypt library which is loaded after file is read.

\n

 "Image"

\n

For ReadFile the two arguments correspond to:

\n

\n

- File Handle

\n

- Memory buffer to store data that is read.

\n

\n

 "Image"

\n

 "Image"

\n

Now to proceed with decryption part some setup is necessary.

\n

\n

- Crypt32.dll is loaded

\n

- CryptStringToBinaryA is called on the data of the certificate.pem file saved in the memory.

\n

\n

 "Image"

\n

CryptStringToBinaryA is used to convert a formatted string into an array of bytes. The flags supplied is in R8 and is 0x1 which means format of the string to be converted is base64. R9 contains the buffer that will receive the output of the function.

\n
 "Image"

\n
The result is the following:

\n
 "Image"

\n
Next setup phase, the malware builds string that will be used later.

\n
 \n
 • ObjectLength
 \n
 • BlockLength
 \n
 • ChainingMode
 \n
 • ChainingModeCBC
 \n

\n
 "Image"

\n
bcrypt.dll module is loaded and proceeds to get addresses for these functions and the jmps to them.

\n
 \n
 • BCryptOpenAlgorithmProvider
 \n
 • BCryptGetProperty
 \n
 • BCryptSetProperty
 \n
 • BCryptGenerateSymmetricKey
 \n
 • BCryptDecrypt
 \n
 • BCryptDestroyKey
 \n
 • BCryptCloseAlgorithmProvider
 \n

\n
 "Image"

\n
First call of the chain is of course BCryptOpenAlgorithmProvider with "AES" supplied as argument. All these functions are from Microsoft DLL so can be searched on official docs for more information.

\n
 "Image"

\n
Second call makes use of the initial strings saved at the first setup stage. It calls on BCryptGetProperty for:

\n
 \n
 • ObjectLength
 \n

- BlockLength -> blocks of 4 bytes
 \n
- ChainingMode
 \n
- ChainingModeCBC
 \n

\n

After the first one, some memory is allocated. After the second, strlen is called to get size of the payload that needs to be decrypted but reads only until first null byte so 17 bytes. Also, VirtualAlloc() is called again and the first 10 bytes of the initial payload are written to it.

\n

 "Image"

\n

 "Image"

\n

 "Image"

\n

strlenw() is called to count "ChainingModeCBC" string. This is necessary for the next call, which is BCryptSetProperty(). The arguments relevant are:

\n

 \n


- szProperty -> "A pointer to a null-terminated Unicode string that contains the name of the property to set." In this case "ChainingMode"
 \n
- pblInput -> "The address of a buffer that contains the new property value. The cbInput parameter contains the size of this buffer." In our case this would be "ChainingModeCBC" with size 15 calculated in the step before.

 \n

\n

 "Image"

\n

 "Image"

\n

Next call is to get size of the key:

\n

 "Image"

\n

symmetric key is generated with BCryptGenerateSymmetricKey(). The fifth argument is the actual key and is passed as reference onto the stack.

\n

 "Image"

\n

Now, the interesting part BCryptDecrypt() is called. This is responsible for decrypting the shellcode. Let's look at the arguments in the debugger. We can use the Microsoft documentation.

\n

 "Image"

\n

First 4 args are passed in: RDX,RCX,R8,R9. The rest are passed as reference onto the stack. The arguments we are interested in are:

\n

 \n


- pblInput The address of a buffer that contains the ciphertext to be decrypted. Second argument.
 \n
- pblIV The address of a buffer that contains the initialization vector (IV) to use during decryption. Fifth argument.

 \n

- pbOutput The address of a buffer to receive the plaintext produced by this function. Seventh argument.

\n

\n

 "Image"

\n

Second Argument (RCX) contains the certificate.pem data that was read into memory before.

\n

The Fifth argument has the IV. This is on the stack, and it is 10 bytes according to the sixth argument.

\n

 "Image"

\n

The pbOutput is null in this case, so no output found.

\n

Following this VirtualAlloc is used to allocate a memory buffer and BCryptDecrypt() is called once again, this time the pbOutput is a pointer to the newly allocated memory. In this case the output can be observed, and we can see the decrypted shellcode.

\n

 "Image"

\n

 "Image"

\n

To make memory section which will contain the payload executable, the malware calls on VirtualProtect() onto that section. The call is used to give the section of memory 0x20-> PAGE_EXECUTE_READ. Finally, the malware can jump to the memory location and resume execution!

\n

 "Image"

\n

 "Image"

\n

With this we have finally made it to the first decrypted shellcode!

\n

 "Image"

\n

Since this was pretty long I will continue with a part 2 soon!

\n

References

\n

\n

- <https://bazaar.abuse.ch/sample/bef04e3b2b81f2dee39c42ab9be781f3db0059ec722ae3b5434c2e63512a68/>

\n

- <https://www.unpac.me/results/612d6d2c-c45d-47ba-a2bb-a218ec753d3f>

\n

- <https://twitter.com/Cryptolaemus1/status/1747394506331160736>

\n

- <https://www.proofpoint.com/us/blog/threat-insight/out-sandbox-wikiloader-digs-sophisticated-evasion>

\n

- <https://www.geoffchappell.com/studies/windows/km/ntoskrnl/inc/api/pebteb/peb/index.htm>

\n

- <https://mohamed-fakroud.gitbook.io/red-teamings-doj/shellcoding/leveraging-from-pe-parsing-technique-to-write-x86-shellcode>

\n

\n","renderedFileInfo":null,"shortPath":null,"symbolsEnabled":true,"tabSize":8,"topBannersInfo":

{\"overridingGlobalFundingFile\":false,\"globalPreferredFundingPath\":null,\"repoOwner\":\"VenzoV\",\"repoName\":\"MalwareAnalysisReports\",\"showInvali

```
cloning-and-archiving-repositories/creating-a-repository-on-github/about-citation-
files", "actionsOnboardingTip": null, "truncated": false, "viewable": true, "workflowRedirectUrl": null, "symbols":
{"timed_out": false, "not_analyzed": false, "symbols": [{"name": "Sample
Information", "kind": "section_1", "ident_start": 2, "ident_end": 20, "extent_start": 0, "extent_end": 3341, "fully_qualified_name": "Sample
Information", "ident_utf16": {"start": {"line_number": 0, "utf16_col": 2}, "end": {"line_number": 0, "utf16_col": 20}}, "extent_utf16": {"start":
{"line_number": 0, "utf16_col": 0}, "end": {"line_number": 133, "utf16_col": 0}}, {"name": "ZIP file contents and
\notepad\n", "kind": "section_2", "ident_start": 570, "ident_end": 601, "extent_start": 567, "extent_end": 1310, "fully_qualified_name": "ZIP file contents
and \notepad\n", "ident_utf16": {"start": {"line_number": 19, "utf16_col": 3}, "end": {"line_number": 19, "utf16_col": 34}}, "extent_utf16": {"start":
{"line_number": 19, "utf16_col": 0}, "end": {"line_number": 37, "utf16_col": 0}}, {"name": "Static
Analysis", "kind": "section_2", "ident_start": 1313, "ident_end": 1328, "extent_start": 1310, "extent_end": 1633, "fully_qualified_name": "Static
Analysis", "ident_utf16": {"start": {"line_number": 37, "utf16_col": 3}, "end": {"line_number": 37, "utf16_col": 18}}, "extent_utf16": {"start":
{"line_number": 37, "utf16_col": 0}, "end": {"line_number": 43, "utf16_col": 0}}, {"name": "Information
Gathering", "kind": "section_2", "ident_start": 1636, "ident_end": 1657, "extent_start": 1633, "extent_end": 3341, "fully_qualified_name": "Information
Gathering", "ident_utf16": {"start": {"line_number": 43, "utf16_col": 3}, "end": {"line_number": 43, "utf16_col": 24}}, "extent_utf16": {"start":
{"line_number": 43, "utf16_col": 0}, "end": {"line_number": 133, "utf16_col": 0}}, {"name": "Dynamic
Analysis", "kind": "section_1", "ident_start": 3344, "ident_end": 3360, "extent_start": 3341, "extent_end": 14892, "fully_qualified_name": "Dynamic
Analysis", "ident_utf16": {"start": {"line_number": 133, "utf16_col": 3}, "end": {"line_number": 133, "utf16_col": 19}}, "extent_utf16": {"start":
{"line_number": 133, "utf16_col": 0}, "end": {"line_number": 411, "utf16_col": 0}}, {"name": "Getting Kernel32.dll &
GetProcAddress", "kind": "section_3", "ident_start": 3368, "ident_end": 3405, "extent_start": 3364, "extent_end": 6457, "fully_qualified_name": "Getting
Kernel32.dll & GetProcAddress", "ident_utf16": {"start": {"line_number": 136, "utf16_col": 4}, "end":
{"line_number": 136, "utf16_col": 41}}, "extent_utf16": {"start": {"line_number": 136, "utf16_col": 0}, "end": {"line_number": 204, "utf16_col": 0}},
{"name": "String builder and Control flow
manipulation", "kind": "section_3", "ident_start": 6462, "ident_end": 6506, "extent_start": 6457, "extent_end": 8164, "fully_qualified_name": "String
builder and Control flow manipulation", "ident_utf16": {"start": {"line_number": 204, "utf16_col": 5}, "end":
{"line_number": 204, "utf16_col": 49}}, "extent_utf16": {"start": {"line_number": 204, "utf16_col": 0}, "end": {"line_number": 235, "utf16_col": 0}},
{"name": "Thread
manipulation", "kind": "section_3", "ident_start": 8168, "ident_end": 8187, "extent_start": 8164, "extent_end": 9092, "fully_qualified_name": "Thread
manipulation", "ident_utf16": {"start": {"line_number": 235, "utf16_col": 4}, "end": {"line_number": 235, "utf16_col": 23}}, "extent_utf16": {"start":
{"line_number": 235, "utf16_col": 0}, "end": {"line_number": 257, "utf16_col": 0}}, {"name": "Decrypting
Shellcode", "kind": "section_3", "ident_start": 9096, "ident_end": 9116, "extent_start": 9092, "extent_end": 14330, "fully_qualified_name": "Decrypting
Shellcode", "ident_utf16": {"start": {"line_number": 257, "utf16_col": 4}, "end": {"line_number": 257, "utf16_col": 24}}, "extent_utf16": {"start":
{"line_number": 257, "utf16_col": 0}, "end": {"line_number": 403, "utf16_col": 0}},
{"name": "References", "kind": "section_2", "ident_start": 14333, "ident_end": 14343, "extent_start": 14330, "extent_end": 14892, "fully_qualified_name": "R
ferences", "ident_utf16": {"start": {"line_number": 403, "utf16_col": 3}, "end": {"line_number": 403, "utf16_col": 13}}, "extent_utf16": {"start":
{"line_number": 403, "utf16_col": 0}, "end": {"line_number": 411, "utf16_col": 0}}}], "copilotInfo": null, "copilotAccessAllowed": false, "csrf_tokens":
{"VenzoV/MalwareAnalysisReports/branches": {"post": "eO7AqGv51ZbWlTh7YjY9oMtGtbz-1DbR2aSa4sTFxO3XB2rAG3GNclmSx-
avthk60Bh1WvV5WhXU14g8flKrmg"}, "/repos/preferences": {"post": "tTKKxf94AdScHP68l0SxEQjnOAIcvm27QvEMAWoG5V3XIUo3c22F3wJ-
DXOJGq3JOZtdA43B5x8uLvJlvyv44Q"}}, "title": "MalwareAnalysisReports/WikiLoader/WikiLoader notepad.md at main ·
VenzoV/MalwareAnalysisReports"}

```