Malware Technology

Personal Journal on Defensive Cybersecurity, DevOps, and anything that comes to mind

---

Statically Unpacking Shellcode-based PE Loaders

Posted on 2022-10-29 in shellcode, malware • 3122 words • 15 minute read
Tags: shellcode, malware

A Quick History Lesson on Relevant Loader Techniques

The ability to run arbitrary executables and scripts in local or remote process memory is an appealing feature for many malware. Malware frameworks possessing this functionality are much more flexible with it, as operators can integrate new tools (in many cases written by completely different developers) without requiring more development on the core malware framework. This also makes features like Metasploit's `migrate` trivial to implement for many C2 frameworks.

Prior to the release of Monoxgas's sRDI and TheWover's Donut, loaders typically required code to manually map a PE file in memory or else utilize Reflective DLL Injection.

The primary issue with the former technique is the amount of effort that must go into the development of that mapping process, requiring more knowledge of the PE file format and its intricacies and edge-cases (TLS callbacks and SEH being a couple). Because the development work required is time-consuming, loader code developed for an unmanaged

executable (e.g. C/C++) will in many cases lack the support for running managed (C# .NET) PE files.

Reflective DLL Injection was popularized due to the ability to run arbitrary DLLs in remote process memory without the need to drop any files to disk or force the call to LoadLibrary in the target process after its initialization (where most of the calls to LoadLibrary would be expected). Malware authors often copied the original implementation by Stephen Fewer for use in their tools. However, a downside to this approach was that the target DLL to be injected had to be integrated with the reflective loading stub at the time of development, so injecting arbitrary DLLs for execution was non-trivial.

Monoxgas's sRDI (shellcode Reflective DLL Injection) wraps arbitrary DLLs with a reflective loading stub to allow for the easy conversion to shellcode. With this approach, a malware framework need only inject the shellcode generated by sRDI into a remote process and set a thread to execute it. This is much simpler to implement from the perspective of the malware framework developer.

With the release of Donut, TheWover took the technique from sRDI and went a few steps further, allowing the conversion of managed and unmanaged PE files (.NET or native, respectively) and even scripts written in VBScript and JScript to shellcode. At this point in time, defensive techniques for Powershell-based malware (popularized by the likes of Powershell Empire) were becoming more mature, causing a migration to other languages. Tools written in .NET became more popular due to ease of development when compared to C/C++, but were commonly run via Cobalt Strike (a native tool). The issue was that the `execute-assembly` command in CS was limited to running the .NET tools in sacrificial child processes and could not be extended to running them in arbitrary remote processes. Donut allowed the conversion of all these different file formats to shellcode that could be run locally or remotely, as it took care of the instantiation and lifecycle of any necessary runtimes. This means that processes birthed from programs written in unmanaged languages (C/C++, Golang, etc.) could run Donut-generated shellcode constructed from managed (.NET) PE files by simply allocating the memory, writing the shellcode to it, and triggering a thread to execute it.

Payload Extraction with Nutex

Ultimately, all of the shellcode-loader tools generate files that might be extracted from an analyzed malware's embedded resources, or from memory. The embedded payload may be found mapped in memory, but with Nutex, we can determine how it was generated:

- The specific loader
- Export called (if DLL)
- Program Arguments

If the unpacked target memory was simply dumped to disk, these would not be seen.

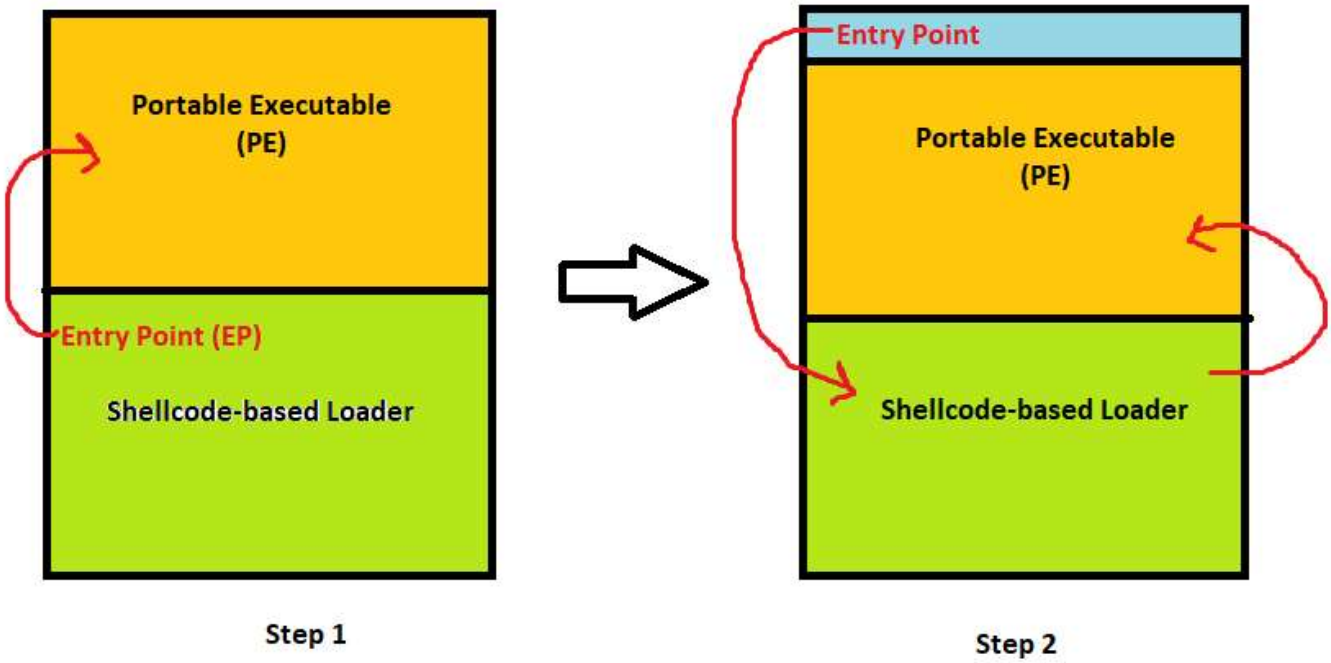
Nutex targets 3 publicly-available, popular shellcode loaders:

- TheWover's Donut
- Monoxgas's sRDI
- Hasherezade's pe2shc

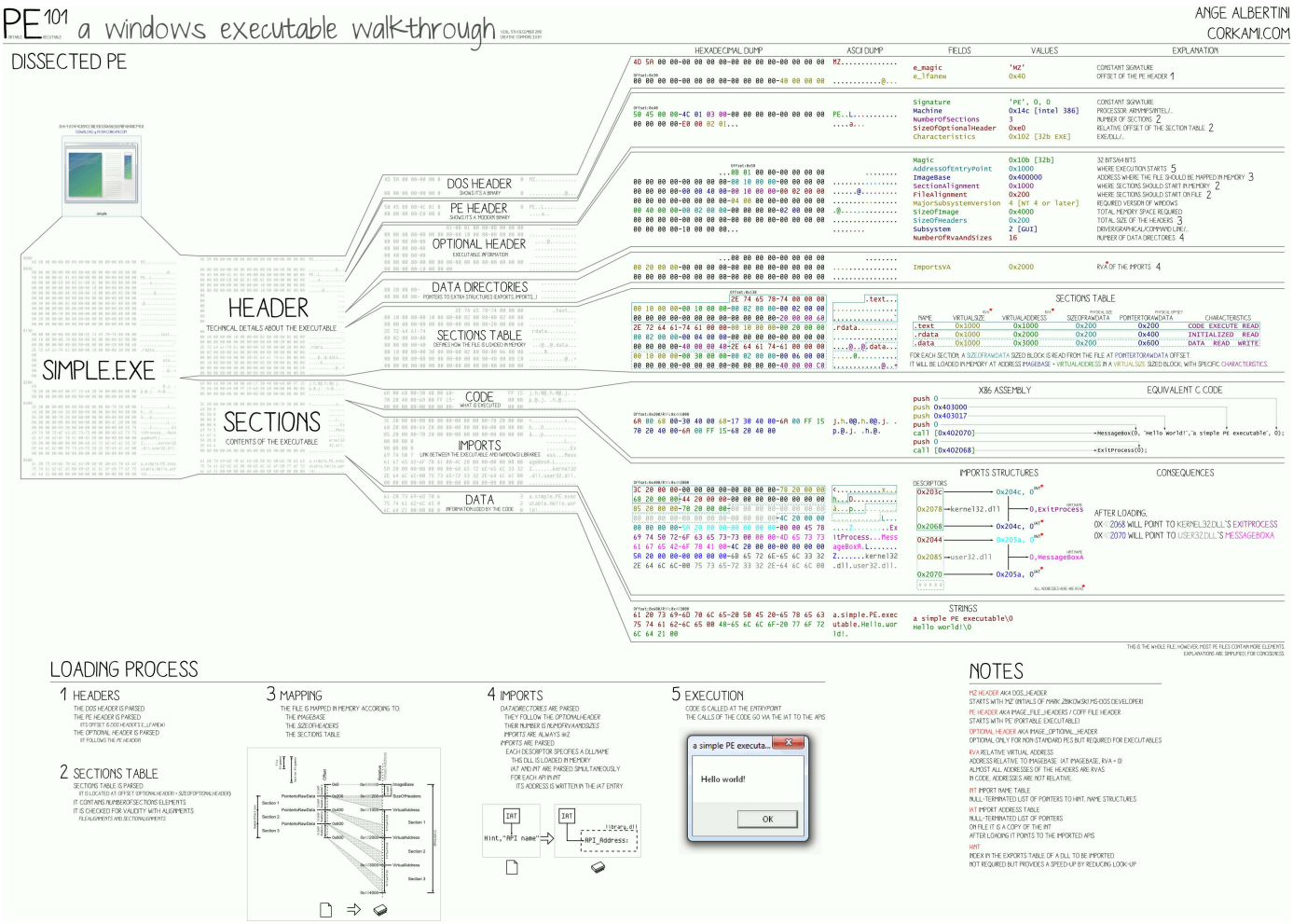
In order to extract the payload, each had to be analyzed in order to determine the necessary steps. We'll start from the simplest to the most complex.

pe2shc

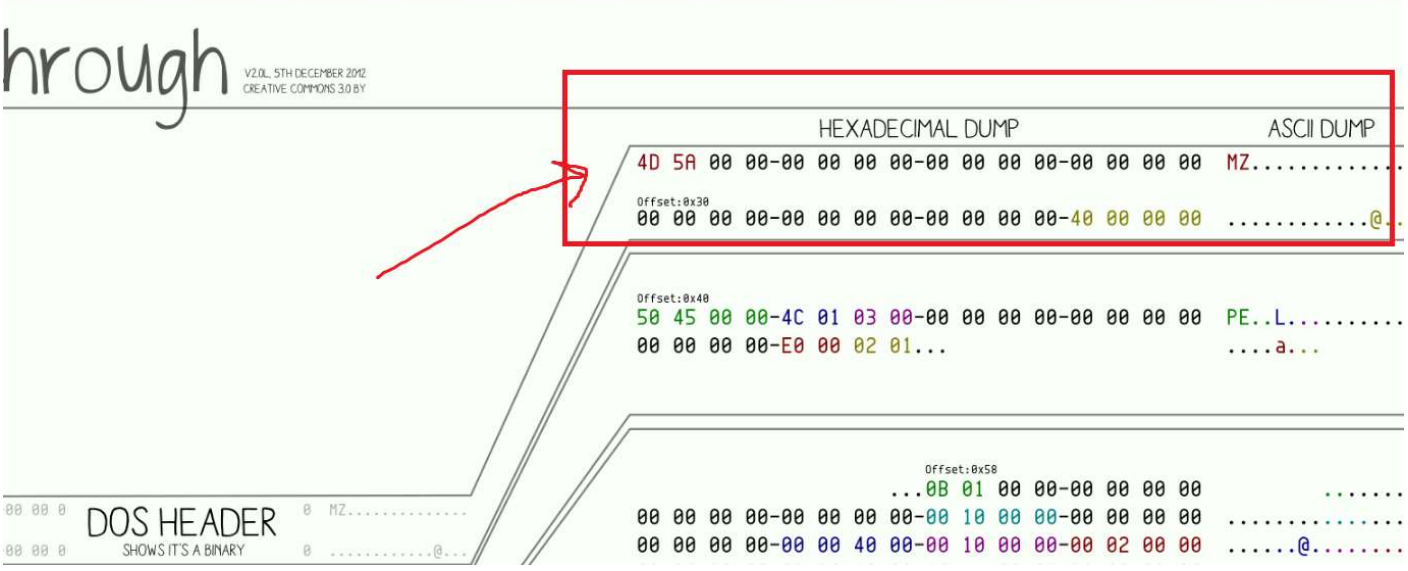
Amongst the three tools targeted, Hasherezade's pe2shc was the simplest to approach.



This is because Hasherezade's tool does not actually functionally modify the input PE file in such a way that it can't be used normally. Portable Executable files contain header fields that can be used to calculate its raw size on disk, so appending data after that calculated offset will not affect normal function.



The beginning of any PE file (EXE, DLL, etc.) contains a magic number (represented in ASCII as MZ) that the operating system will use in part to identify the file as having the PE format. After those two bytes, there are several null bytes that aren't utilized. pe2shc simply repurposes this space to write an initial stub that will redirect code execution to the main stub written after the PE file.



Redirection shellcode stub:

```
bool overwrite_hdr(BYTE *my_exe, size_t exe_size, DWORD raw, bool is64b)
{
	const size_t value_pos = 8;
	size_t redir_size = 0;
	BYTE* redir_code = nullptr;

	BYTE redir_code32_64[] = "\x4D" //dec ebp
		"\x5A" //pop edx
		"\x45" //inc ebp
		"\x52" //push edx
		"\xE8\x00\x00\x00\x00" //call <next_line>
		"\x5B" // pop ebx
		"\x48\x83\xEB\x09" // sub ebx,9
		"\x53" // push ebx (Image Base)
		"\x48\x81\xC3" // add ebx,
		"\x59\x04\x00\x00" // value
		"\xFF\xD3" // call ebx
		"\xc3"; // ret

	BYTE redir_code32[] = "\x4D" //dec ebp
		"\x5A" //pop edx
		"\x45" //inc ebp
		"\x52" //push edx
		"\xE8\x00\x00\x00\x00" //call <next_line>
		"\x58" // pop eax
		"\x83\xE8\x09" // sub eax,9
		"\x50" // push eax (Image Base)
		"\x05" // add eax,
		"\x59\x04\x00\x00" // value
		"\xFF\xD0" // call eax
		"\xc3"; // ret

	BYTE redir_code64[] = "\x4D\x5A" //pop r10
		"\x45\x52" //push r10
		"\xE8\x00\x00\x00\x00" //call <next_line>
		"\x59" // pop rcx
		"\x48\x83\xE9\x09" // sub rcx,9 (rcx -> Image Base)
		"\x48\x8B\xC1" // mov rax,rcx
		"\x48\x05" // add eax,
		"\x59\x04\x00\x00" // value
		"\xFF\xD0" // call eax
		"\xc3"; // ret
```

The main stub will perform the actual loading steps to execute the payload. By doing this, an operator of the tool does not need to set the thread to execute at the offset of the main loader stub, which will be at different addresses for differently-sized PE files. Instead, they can simply set the execution to occur at the starting address of the allocated memory. This redirection loader is specific to the architecture of the input file, with support for both x86 and AMD64 provided.

This can be seen in the `shellcodify` function.

```
BYTE* shellcodify(BYTE *my_exe, size_t exe_size, size_t &out_size, bool is64b)
{
        out_size = 0;
        size_t stub_size = 0;
        int res_id = is64b ? STUB64 : STUB32;
        BYTE *stub = peconv::load_resource_data(stub_size, res_id);
        if (!stub) {
                std::cerr << "[ERROR] Stub not loaded" << std::endl;
                return nullptr;
        }
        size_t ext_size = exe_size + stub_size;
        BYTE *ext_buf = peconv::alloc_aligned(ext_size, PAGE_READWRITE);
        if (!ext_buf) {
                return nullptr;
        }
        memcpy(ext_buf, my_exe, exe_size);                    (1)
        memcpy(ext_buf + exe_size, stub, stub_size);          (2)

        DWORD raw_addr = exe_size;
        overwrite_hdr(ext_buf, ext_size, raw_addr, is64b);    (3)

        out_size = ext_size;
        return ext_buf;
}
```

pe2shc will perform the following steps:

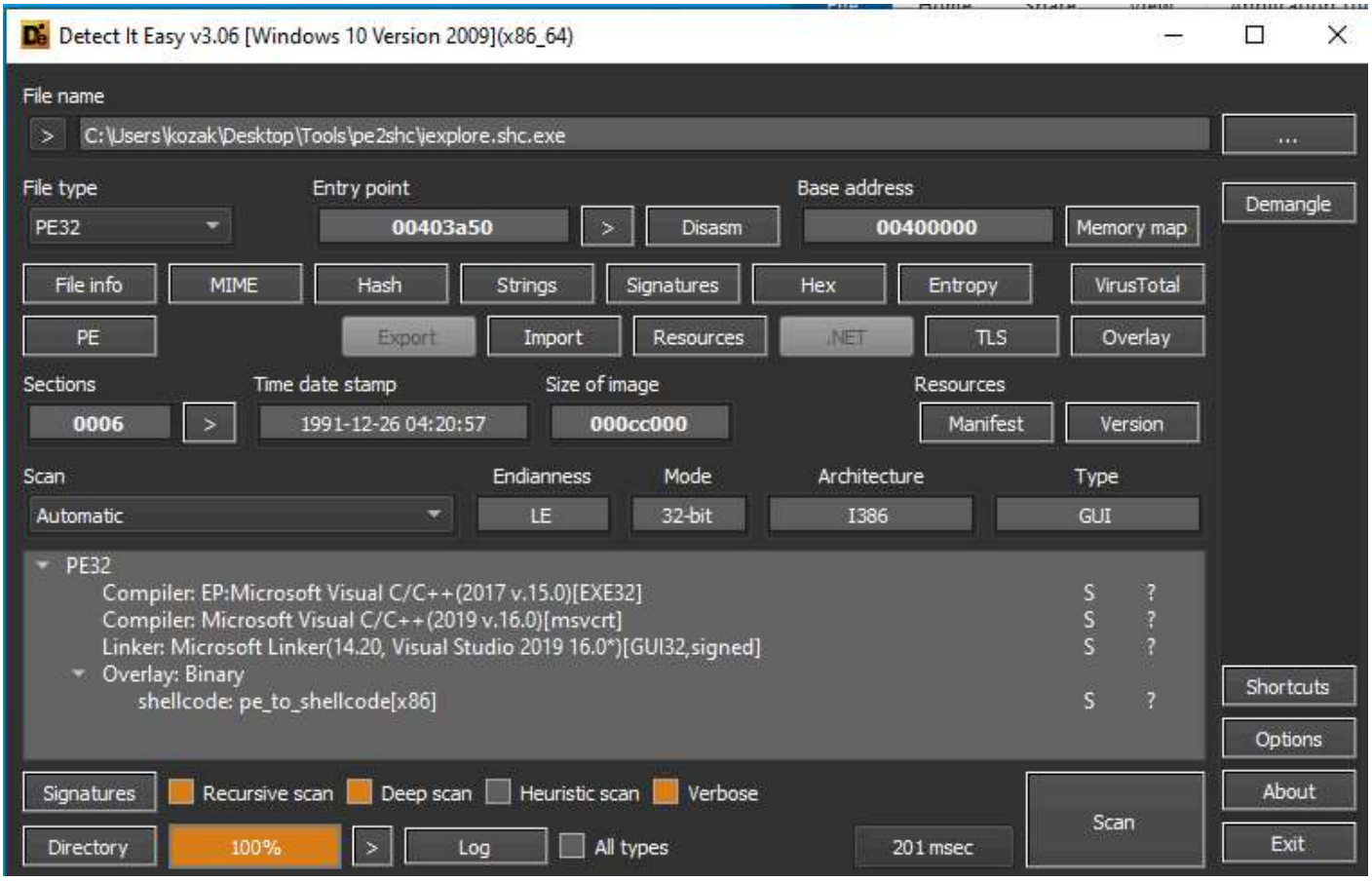
1) Write input PE file to buffer

2) Write main shellcode loader after PE file

3) Overwrite beginning bytes of PE file with the redirection stub that will redirect execution to (2)

Detection

Utilizing the latest code from Detect-It-Easy, we can see that it detects out of the box usage of pe2shc. Because pe2shc does not functionally modify the usage of the PE file, DIE can also detect and parse the input payload (in this case, Internet Explorer).

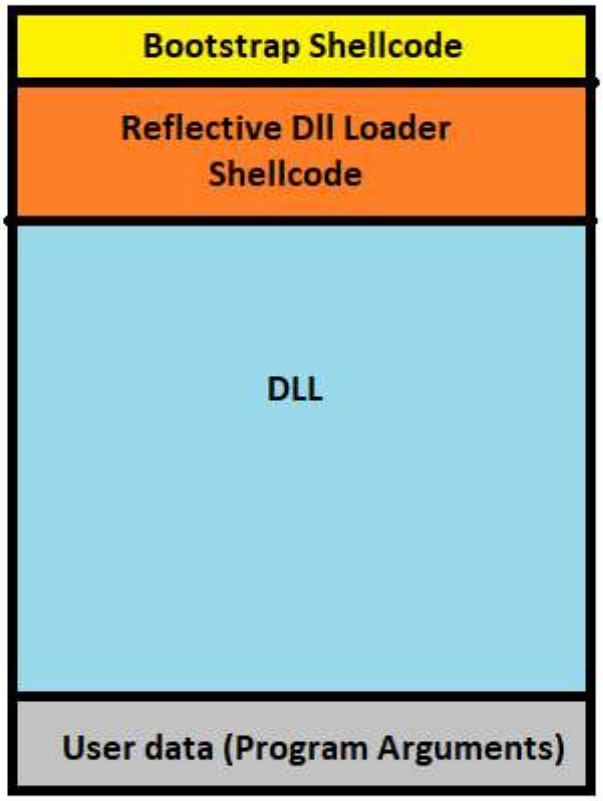


Unpacking

The extra data of the two stubs will not affect the usage of a given PE file modified by pe2shc. As a result, double-clicking or running via the CMD prompt one of the generated files will give the same result as if the input file had not been modified.

No unpacking steps are necessary.

Monoxgas's sRDI



Monoxgas's sRDI takes a slightly different approach than Hasherezade's pe2shc. Firstly, sRDI only takes DLLs as input. It works as an alternative to Reflective DLL Injection, allowing the ability to load arbitrary DLLs, but not regular EXEs.

sRDI supports the ability to specify an export of the DLL to call, as well as optional arguments to pass to that function. Unlike pe2shc, the generated file is not a valid PE file any longer. Like pe2shc, however, two shellcode stubs exist. In sRDI, these stubs are adjacent, starting at the beginning of the file with the bootstrap followed by the RDI loader. The author provides a comment describing this.

```
# Ends up looking like this in memory:
# Bootstrap shellcode
# RDI shellcode
# DLL bytes
# User data
return bootstrap + rdiShellcode + dllBytes + userData
```

The purpose of the bootstrap shellcode is to inform the reflective loader stub of the function to call, any arguments to pass to that function, and any additional flags used to toggle obfuscation features. The reflective loader stub is static, and doesn't change much beyond whatever small differences compiler versions might create inadvertently.

To hide the exported function called in the input DLL, the name is hashed with ROT13. The value is stored in the bootstrap shellcode. The code in the `if` case above the `else` statement appears to be unused.

```
                functionHash = 0

                for b in function:
                        functionHash = ror(functionHash, 13, 32)
                        functionHash += b
```

The location of the user data will be calculated, packed into the shellcode as a 4-byte integer before its length. The location of the user data is used as a pointer and stored in the AMD64 `r8` register. The length of the user data is stored in `r9`. (the `d` in `rd9` just specifies we want to store the value of the user data as the lower 32-bits in the register). In AMD64, the calling convention uses these registers as the 3rd and 4th arguments passed to a function. As a result, they will be passed to the `lpUserData` and `dwUserdataLen` parameters in ShellcodeRDI/ShellcodeRDI.c.

```
ULONG_PTR LoadDLL(PBYTE pbModule, DWORD dwFunctionHash, LPVOID lpUserData, DWORD dwUserdataLen, PVOID pvShellcodeBase, DWORD dwFlags)
{
```

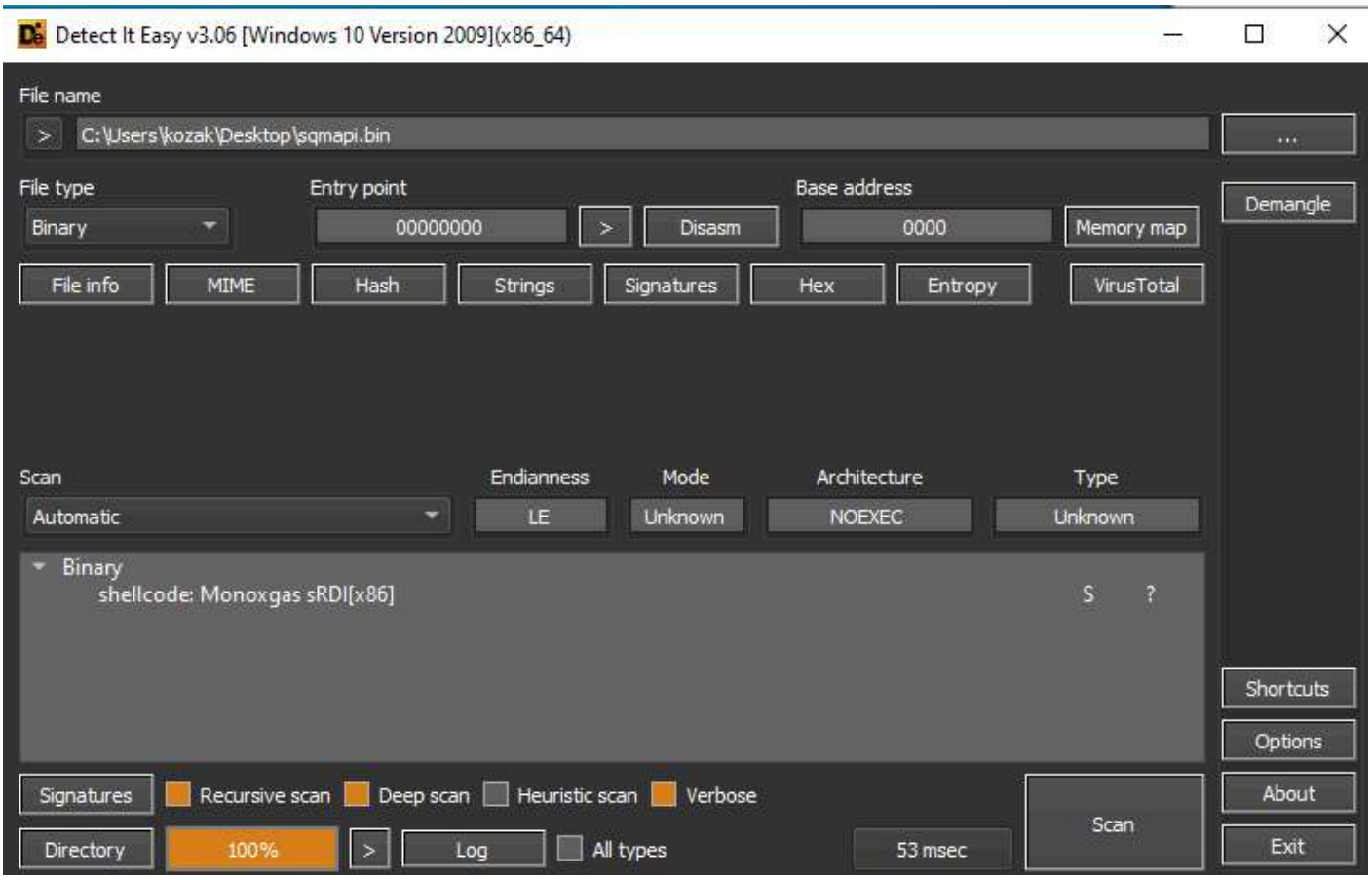
In a similar way, the flags are packed and stored in the bootstrap shellcode such that when the shellcode runs, they will be pushed on the stack to be used by the function shown above. These flags enable minor obfuscation techniques such as clearing the PE header of the loaded DLL to make its file format harder to analyze in memory. Other flags increase the amount of time before each import required by the RDI loader stub is imported, as a measure to try to bypass runtime detections. Finally, the base address of the bootstrap shellcode could be passed to DLL function called.

```
    if arguments.clear_header:
        flags |= 0x1

    if arguments.obfuscate_imports:
        flags = flags | 0x4 | arguments.import_delay << 16

    if arguments.pass_shellcode_base:
        flags |= 0x8
```

In essence, you can think of the sRDI output (as well as Donut) shellcode as a really funky file format. By using known offsets, we can unpack files that used sRDI out-of-the-box.

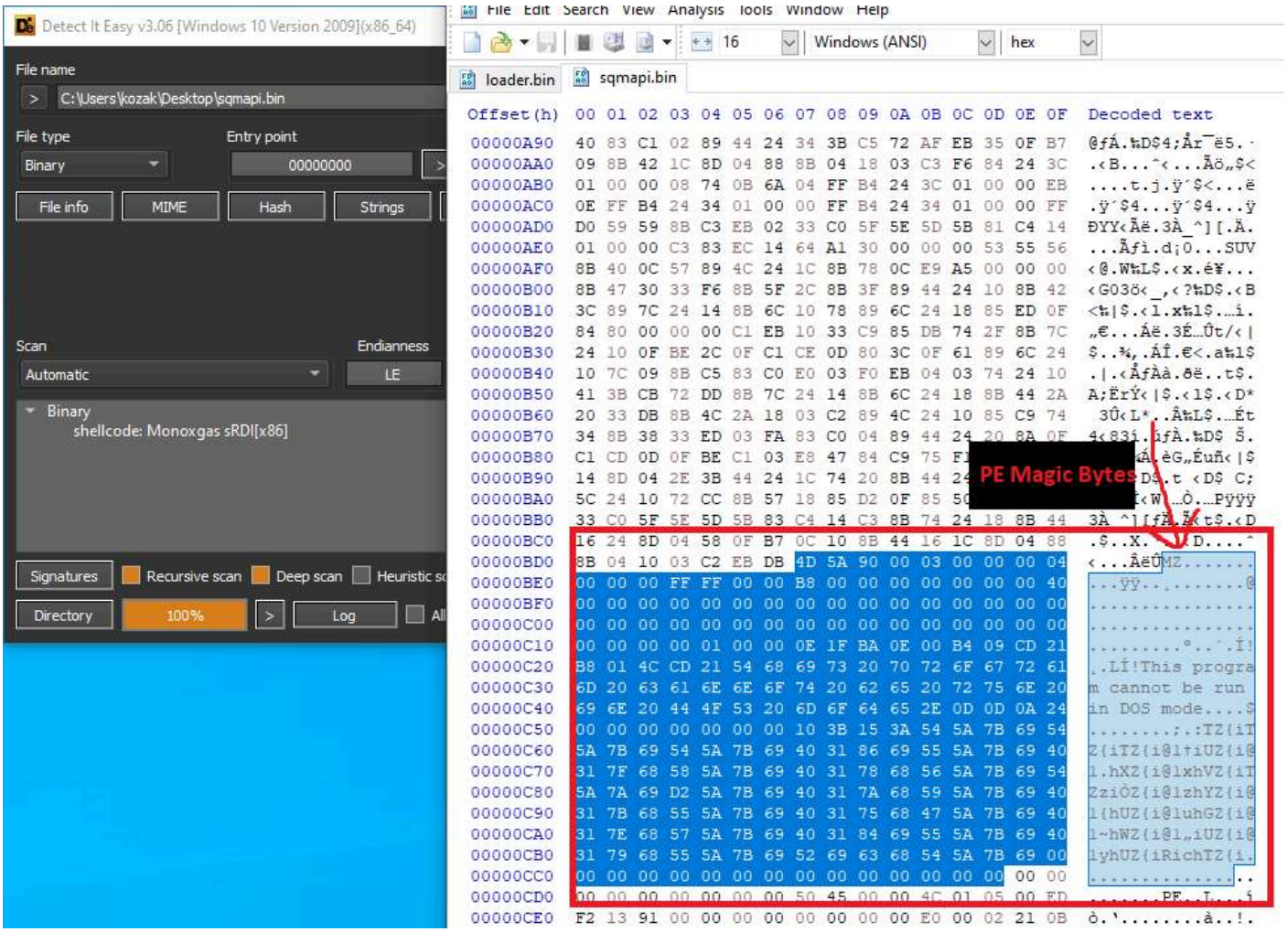
Detection

As with pe2shc, I wrote a Detect-It-Easy detection for sRDI.



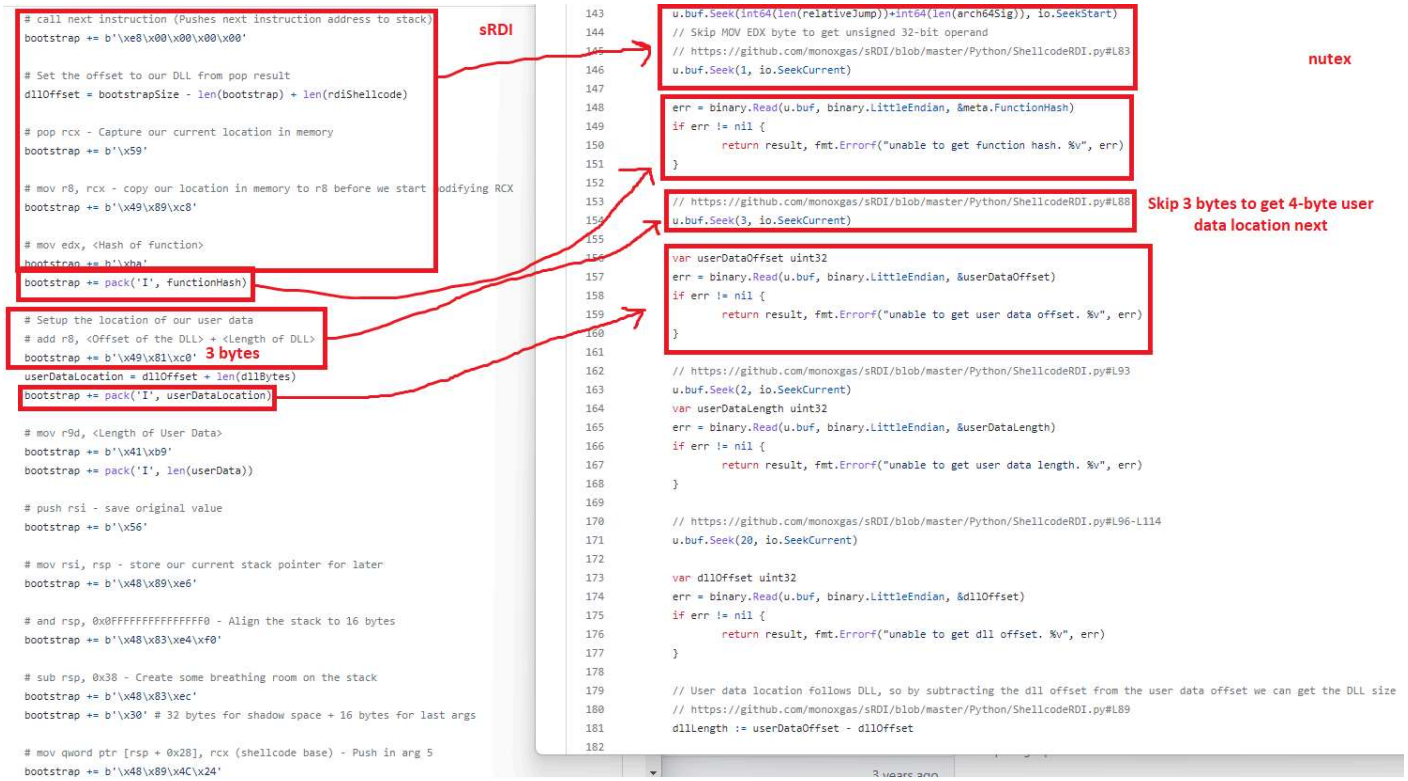
Note that this will work with some other implementions of sRDI if the same bootstrap shellcode format is used. I noticed this in some cases with implementations written in other languages like this.

Unpacking



To get the data we want, we need to extract from the bootstrap shellcode the following values:

- DLL Length
- DLL Offset
- Flags
- User Data Length
- User Data Location
- Target Function Hash



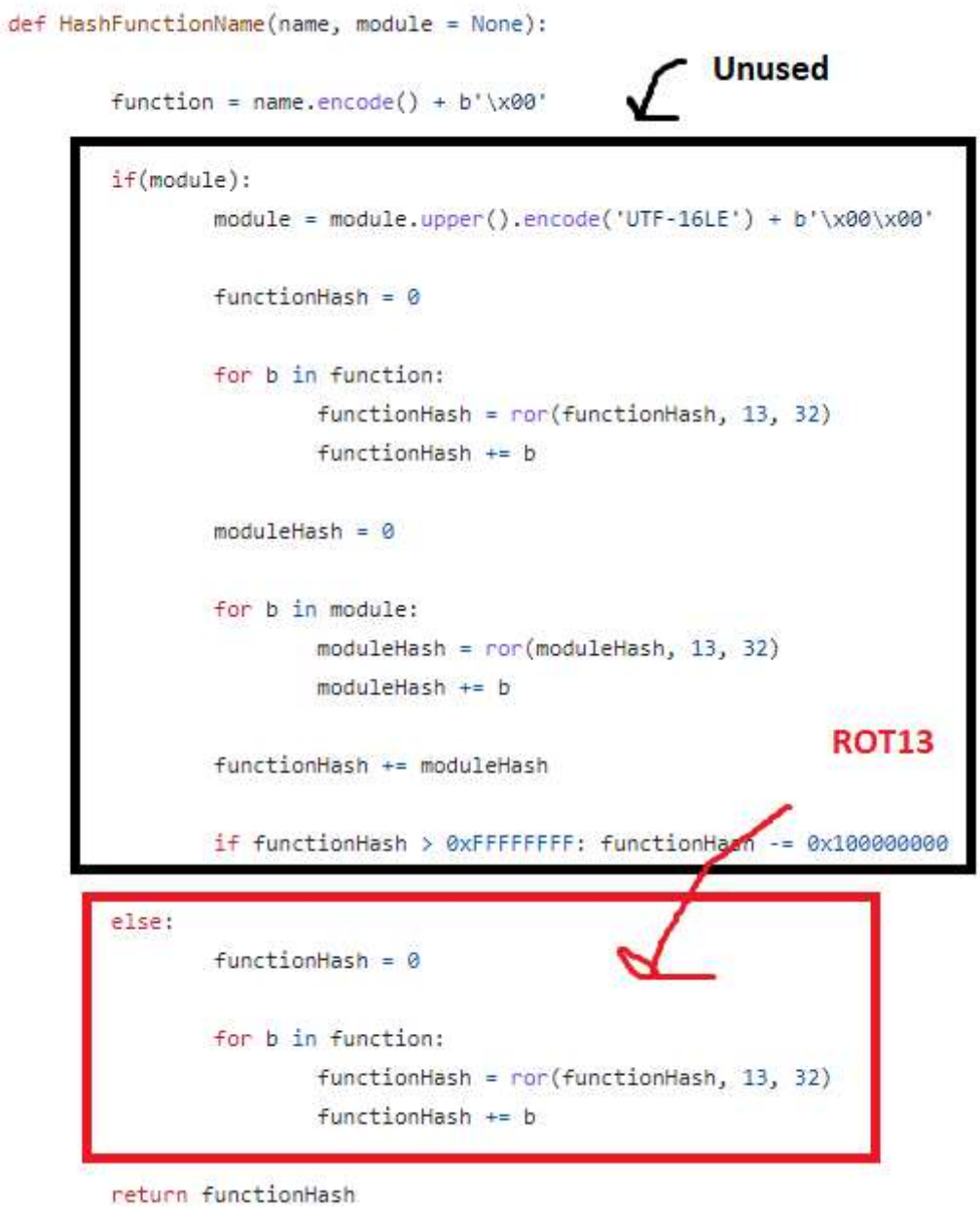
Essentially, we read past known offsets to extract the data being packed with the `pack` Python calls. By observing the arguments passed to the `pack` function, we can identify the number of bytes to read. With this information, we can separate the embedded DLL, flags, and user data from the file! Unpacking sRDI is fairly straight-forward.

`There is a key weakness to the design of the nutex unpacker!` This unpacker is intended only to extract payloads and ancillary information from files generated using out-of-the-box versions of these packers. An attacker with sufficient knowledge can modify the assembly just slightly in order to break nutex. The use case of nutex is to catch those without that level of knowledge, or tools like Sliver that use the Donut packer as a feature of an entire malware framework.

An interesting thing to note is that the input DLL and user-data is actually not encrypted in the packed output of sRDI. As a result, a simple script to find a valid PE header and extracting the payload using the relevant structure fields might suffice. With the user data (program arguments) left in plaintext as well, a simple check for the default user data of dave (yes, really) in the last 4 bytes of a region might yield some results in flagging some memory regions as malicious.

However, these processes would not yield the function called within the DLL. If the DLL contains many exported functions as a form of obfuscation (to hide from the malware analyst which one(s) is used), then simply extracting the payload would not be enough. It was required to reimplement the 32-bit ROT13 implementation in order to perform the same hashing function; this would allow us to test equivalence between what is set by the bootstrap shellcode, and the various results when using DLL function export names as input. As a result of comparing the stored hash value with the hash values produced

by running the DLL export names through the function, we can determine the actual target function.



The value of the function shown above is calculated and then passed to the function that builds the bootstrap shellcode.

```go
func (p *Payload) TargetFunction() (string, error) {
	fmt.Printf("[*] DLL Size (bytes): %d\n", len(p.DLL))
	fmt.Printf("[*] First 5 bytes: %x\n", p.DLL[:5])
	dll, err := pe.NewBytes(p.DLL, &pe.Options{})
	if err != nil {
		return "", fmt.Errorf("unable to open embedded DLL. %v", err)
	}
	err = dll.Parse()
	if err != nil {
		return "", fmt.Errorf("unable to parse embedded DLL. %v", err)
	}

	fmt.Printf("[*] Target Function Hash: 0x%x\n", p.Metadata.FunctionHash)

	fmt.Printf("[*] Number of Exports: %d\n", len(dll.Export.Functions))
	for _, export := range dll.Export.Functions {
		var funcHash uint32
		cstr := bytes.NewBufferString(export.Name).Bytes()
		cstr = append(cstr, 0x00) // add null byte to match c-string
		for _, char := range cstr {
			// https://github.com/monoxgas/sRDI/blob/9fdd5c44383039519accd1e6bac4acd5a046a92c/Python/ShellcodeRDI.py#L5
			// 13 is a constant for rotation
			funcHash = bits.RotateLeft32(funcHash, -13) // rotating left by negative number makes it rotate right
			funcHash += uint32(char)
		}
		if funcHash == p.Metadata.FunctionHash {
			return export.Name, nil
		}
	}
	return "", fmt.Errorf("no matching exported function for hash: 0x%x\n", p.Metadata.FunctionHash)
}
```

Read and Validate PE format

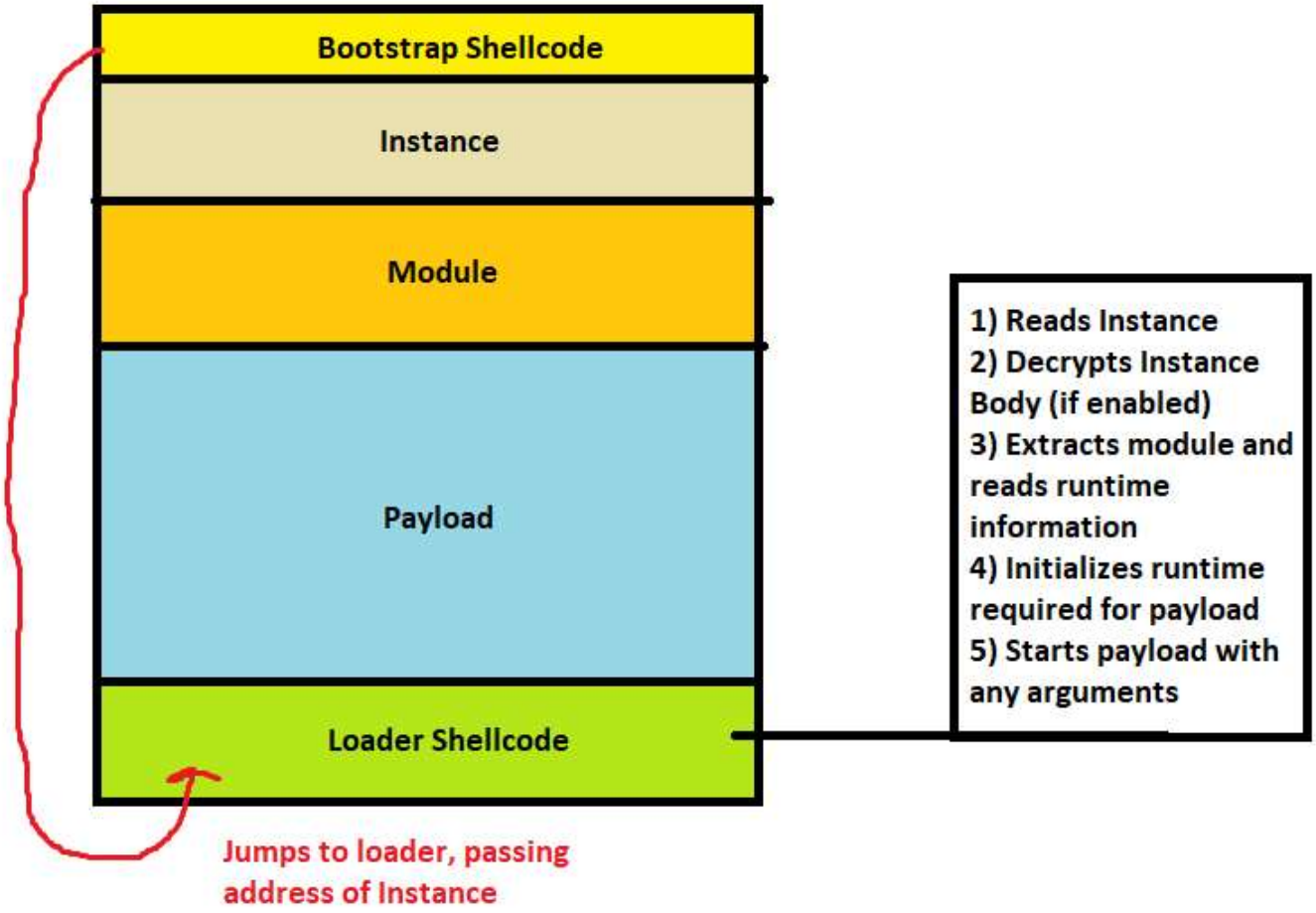
Loop over every function exported from DLL

Calculate ROT13 hash for export function name

Check if hashed value == hash value extracted from bootstrap

In essence, nutex brute-forces the hash-value extracted from the bootstrap shellcode using the function names exported from the DLL. When one of the hash values calculated for a function matches, we know that that is the one called by the RDI loader stub.

TheWover's Donut



Donut was more complex than the other two shellcode loaders by orders of magnitude. That is due in major part to the fact that Donut supports so many more features than the others, such as different payload types.

Payload types supported by Donut include:

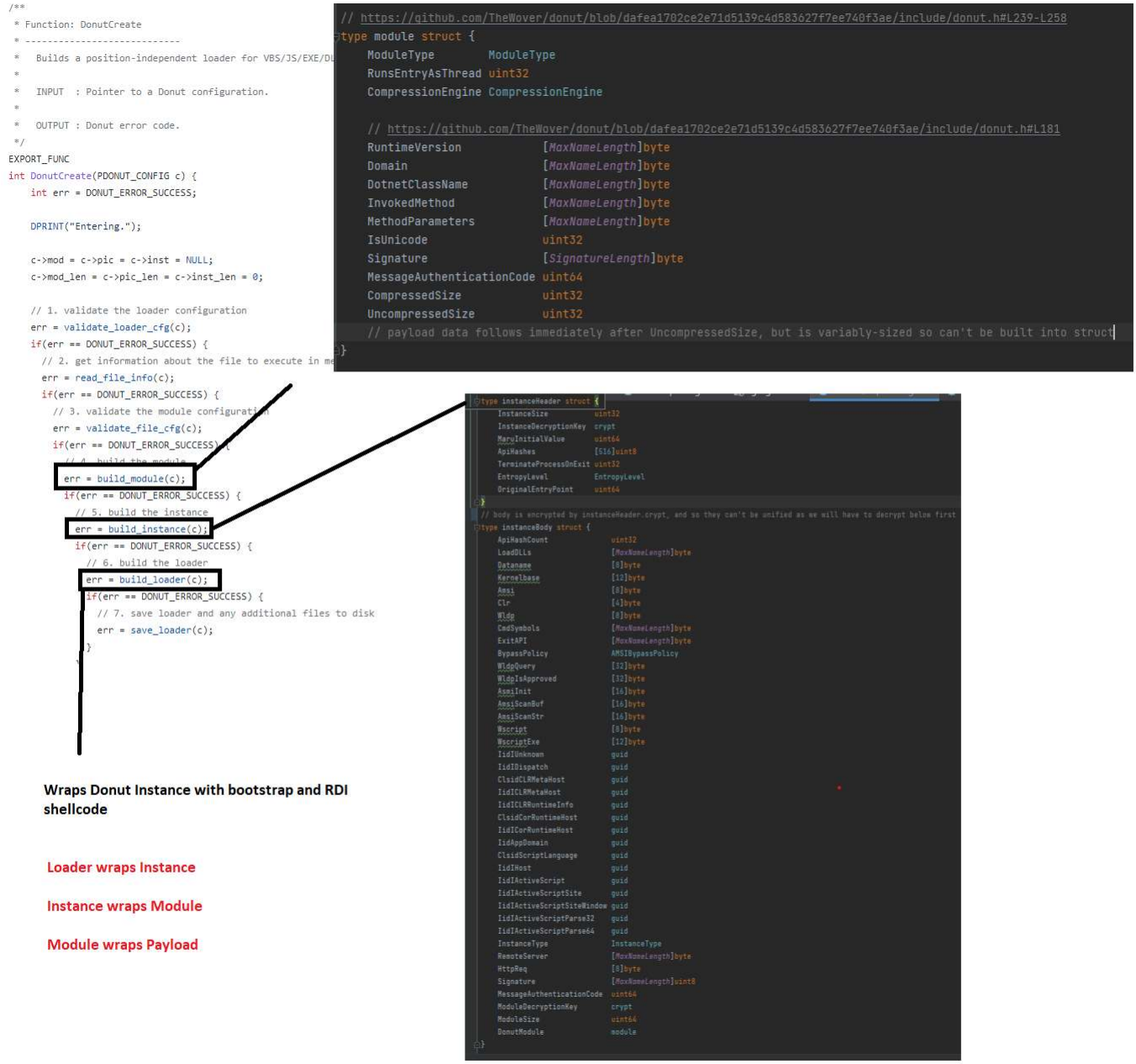
- Native/Managed DLLs (x86/AMD64)
- Native/Managed EXEs (x86/AMD64)
- JScript
- VBScript

Multiple compression algorithms are supported to keep the final file generated smaller. Variably-sized parts of embedded structures alongside optional (but set by default) encryption make unpacking Donut more like peeling back layers of an onion.

The final output of Donut can be summed up as the whole of a boostrap shellcode, a Donut “instance”, and a reflective loader stub shellcode. Conceptually, this isn’t too much different than the other shellcode loaders studies. They all contain the basics of

a boostrap at the beginning, a payload somewhere in the middle, and the loader stub at the end that performs the main logic. Different terminology might be used, such as pe2shc calling its bootstrap a “redir” (as in redirection) stub, but the concepts are the same.

Where this becomes more complicated is in the complexity of the instance+module combination that precedes the user inputted payload.



Disregarding the embedded Module, the Donut Instance contains all the fields necessary for the reflective loader to setup any required runtimes for the different payload types, flags and associated data for bypassing AMSI (relevant for JScript/VBScript payloads), remote HTTP servers for payload retrieval (for the HTTP module type), and the decryption key for instance body data (including the embedded module) if enabled.

The module size is stored so that the reflective loader stub can know how much data to read after the instance to continue the unpacking process.

There are two types of instances currently supported: those with embedded modules (1), and those that need to retrieve their modules from remote HTTP servers (2). References are made and a constant is defined for a third module type (retrievable over DNS). But as Donut was released several years prior with development stalled, this is probably unlikely to be supported. Embedded modules are the default, and will simply append the user inputted payload after the module header struct.

If compression is enabled, the user inputted payload will first be compressed before appending it after the module struct. Several types of compression algorithms are supported at this time:

- aPLib
- LZNT1
- Xpress
- XpressHuffman

The type of decompression the loader should use is set in the module struct.

Looking at the relevant code within Donut, all but aPLib are utilized via the Windows API within the loader.

```
DPRINT("Decompressing with RtlDecompressBuffer(%s)",
  mod->compress == DONUT_COMPRESS_LZNT1 ? "LZNT" :
  mod->compress == DONUT_COMPRESS_XPRESS ? "XPRESS" : "XPRESS HUFFMAN");

nts = inst->api.RtlDecompressBuffer(
      (mod->compress - 1) | COMPRESSION_ENGINE_MAXIMUM,
      (PUCHAR)unpck->data, mod->len,
      (PUCHAR)&mod->data, mod->zlen, &len);

//_VirtualFree(ws, 0, MEM_RELEASE | MEM_DECOMMIT);

if(nts == 0) {
  // assign pointer to mod
  mod = unpck;
} else {
  DPRINT("RtlDecompressBuffer failed with %"PRIX32, nts);
  goto erase_memory;
}
```

This is great for us, as it makes the decompression trivial, as we can call the Windows API from Golang in the same way. Within Donut, aPLib was utilized via a static library stored within the repository, so I ended up utilizing a third-party implementation in GoLang to do the same.

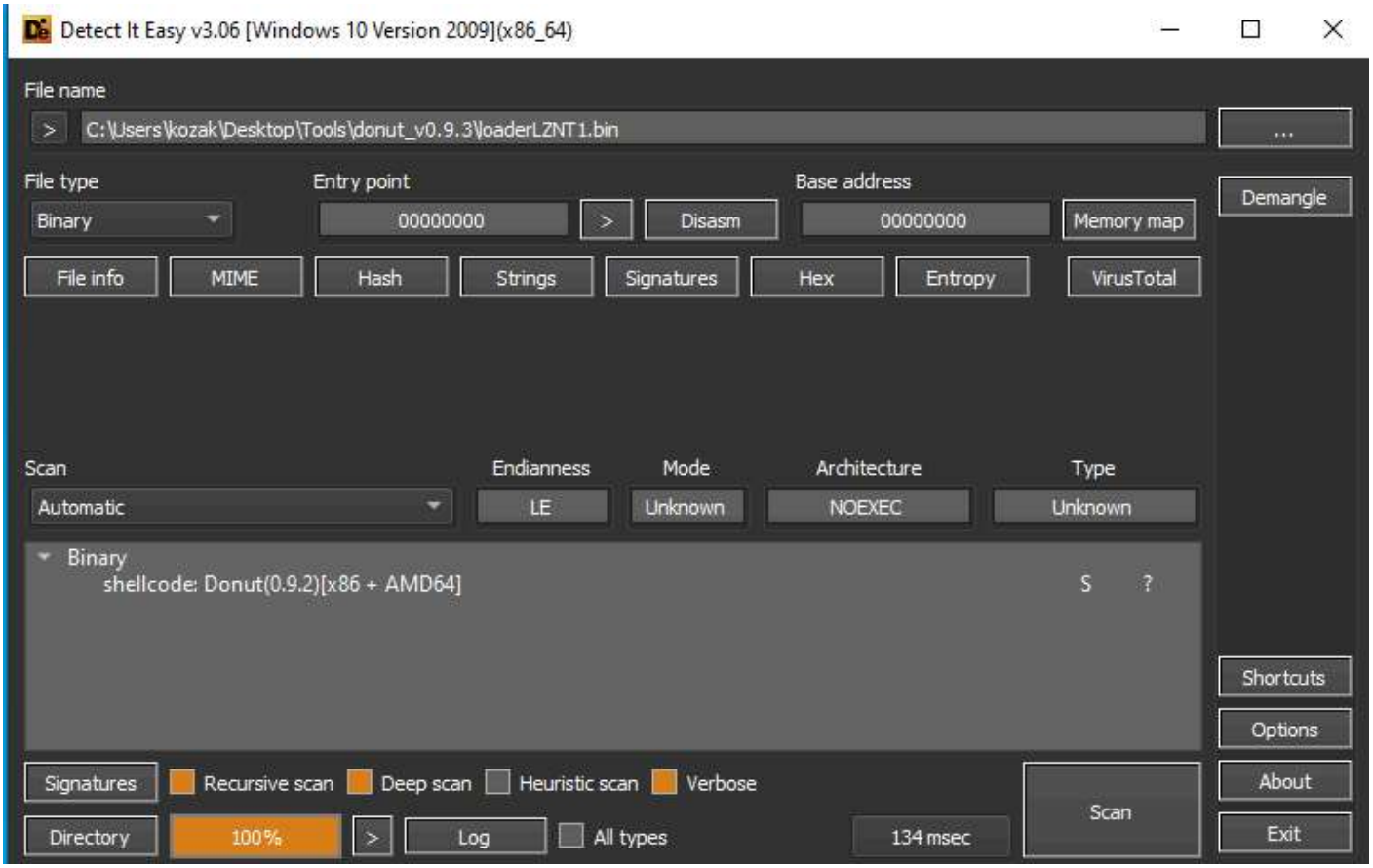
Encryption is supported via the default entropy option. The other entropy option (besides no entropy) is the DONUT_ENTROPY_RANDOM option. This is really only applicable to .NET payloads, as it will generate a random name for the AppDomain for the CLR runtime it instantiates. This can be important for evading some detections looking for known-malicious or suspicious AppDomains. With the default entropy option enabled, the instance body, along with the module struct and the embedded payload (if it exists) are encrypted with the symmetric Chaskey algorithm using a random key that is stored in the plaintext portion of the Instance. Starting with the `api_cnt` field and everything within the Instance struct thereafter will be encrypted.

The function (for DLLs) and parameters to be passed to the embedded payload when loaded are stored in the module header.

Many other fields are present within the Instance and Module, but these are the most pertinent for detecting the type of payload, arguments passed, and encryption/compression we must deal with to unpack the payload.

Detection

My detection written for Detect-It-Easy will detect payloads generated by Donut, as well as the CPU architecture for which they were generated.



Unpacking

Unpacking Donut is more complicated than the other loaders due to encryption and compression within the embedded structs. With so many fields present within Donut's Instance and Module, approaching the problem in the same way as I had done with sRDI would not be feasible. In that approach, I simply used known offset to skip bytes and read the DWORDs that were relevant. The larger number of fields within the Donut Instance and Module would make this cumbersome. A different approach was needed.

In many cases, having a larger number of fields wouldn't be a problem, as Golang's binary.Read can read data from a byte stream into a primitive or struct with a predictable size. This could not be done with Donut payloads due to a theoretically unlimited payload size and encryption (under default conditions) of the field containing its size.

As a result, I had to break up the Instance into two parts: a plaintext header and a (potentially encrypted) body.

I start by reading the entire instance, using known offsets in the bootstrap shellcode (left) to get the size and then the instance itself.

```
// call $ + c->inst_len
PUT_WORD(pl,  c->inst_len);
PUT_BYTES(pl, c->inst, c->inst_len);
// pop ecx
PUT_BYTE(pl,  0x59);

// x86?
if(c->arch == DONUT_ARCH_X86) {
  // pop edx
  PUT_BYTE(pl, 0x5A);
  // push ecx
  PUT_BYTE(pl, 0x51);
  // push edx
```

```
377
378     // https://github.com/TheWover/donut/blob/dafea1702ce2e71d5139c4d583627f7ee740f3ae/donut.c#L1236
379     // despite saying PUT_WORD, this actually puts 4 byte integer of offset to loader AFTER current offset
        // this means from after offset 5 (1 byte rel jump + 4 byte instance length) + instance length
381     var instanceLengthBytes uint32
382     err = binary.Read(u.buf, binary.LittleEndian, &instanceLengthBytes)
383     if err != nil {
384             return nil, fmt.Errorf("unable to read instance length. %v", err)
        }
386
387     donutInstanceBytes := make([]byte, instanceLengthBytes)
388     err = binary.Read(u.buf, binary.LittleEndian, &donutInstanceBytes)
389     if err != nil {
390             return nil, fmt.Errorf("unable to read donut instance. %v", err)
391     }
```

That data includes the instance header and body (which includes the module). As the instance header is a known size, I can then read that into a structure. With the plaintext instance header, I get the instance decryption key, which then allows me to decrypt the body and module using the same library as Sliver does to generate a payload it can inject into remote processes. Since Chaskey is a symmetric encryption algorithm, I am able to use the same function that is used to encrypt data to decrypt data given the key extracted from the Donut instance header.

With the size of the module now found in the decrypted instance body, and with the rest of the Module containing fixed offsets, I am able to calculate the size of the payload and extract it from the byte stream. An important thing to note is that this won't work with HTTP modules. With HTTP modules, no payload data is stored in the generated artifact; rather, fields within the encrypted instance body inform the reflective loader where to download it from.

With the instance header, decrypted instance body, module header, and payload, we now have everything we need to extract not only the payload but any options (export function, type of payload, arguments passed) used when the artifact was generated!

Limitations of Nutex

To build and use `nutex` for yourself, you can find the source on GitHub!

```
Be aware that this tool only works on uses of pe2shc, sRDI, and Donut that used out-of-the-box

Enjoy!
```

Find me around the web:
GitHub

Built with Hugo, using the theme smigle, which was influenced by the theme smol.