

# WarbirdModern.md

---

[github.com/WitherOrNot/warbird-docs/blob/main/WarbirdModern.md](https://github.com/WitherOrNot/warbird-docs/blob/main/WarbirdModern.md)

---

## Modern Warbird

---

By WitherOrNot

## Introduction

---

Warbird in modern Windows is a surprisingly shallow system, with relatively few tricks that are actually in use. For this writeup, only the obfuscation methods that have been observed in actual binaries will be discussed for brevity.

## Encryption Segments

---

Warbird encrypts and decrypts sets of functions, known as "segments", collectively at runtime. Functions contained in particular segments are not necessarily related, and often functions from one segment will reference functions in another segment. The main purpose of this system is to prevent dumping all encrypted functions at once through runtime unpacking.

Within binaries that have encryption segments, like `sppsvc.exe`, there are multiple sections named `?g_Encry`, and each contains an `ENCRYPTION_SEGMENT` struct describing the segment as follows:

```

typedef struct _FEISTEL_ROUND_DATA {
    DWORD round_function_index; // Index of function used for round
    // Parameters for round functions
    DWORD param1;
    DWORD param2;
    DWORD param3;
} FEISTEL_ROUND_DATA;

typedef struct _ENCRYPTION_BLOCK {
    DWORD flags;
    DWORD rva;
    DWORD length;
} ENCRYPTION_BLOCK;

struct PRIVATE_RELOCATION_ITEM
{
    ULONG rva: 28;
    ULONG reloc_type: 4;
};

typedef struct _ENCRYPTION_SEGMENT {
    BYTE hash[32]; // SHA-256 hash of all segment data that follows
    ULONG segment_size;
    ULONG _padding0;
    ULONG segment_offset;
    ULONG reloc_table_offset; // Offset to a global table of PRIVATE_RELOCATION_ITEMS
    ULONG reloc_count; // Length of table
    ULONG _padding1;
    UINT64 preferred_image_base; // Preferred base address of PE as specified in headers
    ULONG segment_id; // Internal segment ID as described in Warbird configuration
    ULONG _padding2;
    UINT64 key; // Feistel cipher key
    FEISTEL_ROUND_DATA rounds[10];
    DWORD num_blocks; // Length of blocks array
    ENCRYPTION_BLOCK blocks[1]; // Blocks of data within module to be decrypted
} ENCRYPTION_SEGMENT;

```



During runtime, encryption segments are decrypted by a system call using [NtQuerySystemInformation](#). The arguments provided for this call are contained in the following struct:

```
typedef struct _ENCRYPTION_SEGMENT_ARGS {
    UINT64 operation; // 1 (decrypt) or 2 (re-encrypt)
    PVOID segment; // Pointer to ENCRYPTION_SEGMENT
    PVOID base; // Base address of module
    UINT64 preferred_base; // Same as segment preferred_image_base
    PVOID reloc_table; // Address of the same table referred to by reloc_table_offset
    UINT64 reloc_count; // Number of relocations in global table
} ENCRYPTION_SEGMENT_ARGS
```



and the system call to decrypt the segment is done like so, with **operation** set to **1**:

```
NtQuerySystemInformation(0xB9, &segment_args, sizeof(segment_args), 0);
```



After the system call, the encrypted data is decrypted in-place, and the binary is able to jump or call to any formerly encrypted functions as usual. Once the encrypted functions are no longer needed, another system call is done with **operation** set to **2** to re-encrypt the segment.

## Heap Executes

---

While encryption segments may deter basic static analysis, they may be easily defeated by placing a breakpoint within the encrypted code, waiting for it to be called, dumping the process from memory, and piecing together the decrypted segments. Thus, for more sensitive functions, another method of code encryption is used, called "heap execute". This method allocates decrypted code in the process heap and jumps to it, preventing simple methods of breakpointing and disrupting attempts at control flow analysis or code tracing.

Heap executes use structs typically placed together near the beginning of the **.text** section, described as follows:

```
// Struct is aligned to 8 bytes
typedef struct _HEAP_EXECUTE {
    BYTE hash[32];
    ULONG hexec_size;
    ULONG virt_stack_limit;
    ULONG hexec_offset:28;
    ULONG _padding0;
    ULONG checksum:8; // Unused
    ULONG unused0:8; // Unused
    ULONG rva:28;
    ULONG size:28;
    ULONG unused1:28; // Unused
    ULONG unused2:28; // Unused
    UINT64 key;
    FEISTEL_ROUND_DATA rounds[10];
} HEAP_EXECUTE;
```



Each **HEAP\_EXECUTE** provides only 1 function to be decrypted. Similar to encryption segments, heap executes are invoked with a call to **NtQuerySystemInformation**, using the following arguments struct:

```
typedef struct _HEAP_EXECUTE_ARGS {
    UINT64 operation; // 3 (heap execute)
    PVOID heap_execute; // Pointer to HEAP_EXECUTE
    PVOID return_val; // Pointer to variable that will hold return value
    ULONG arguments[1]; // Unbounded array of arguments to be passed to function
}
```



The system call is the same as with encryption segments. Once called, process execution will jump to the decrypted code in the heap.

Letting **rip** be the address of the start of the decrypted code, the heap layout is as follows:

```
rip - 0x10: Call offset
rip - 0x08: NtQuerySystemInformation syscall number (usually 0x36)
rip: Decrypted code
```



Letting **rsp** be the stack pointer, the stack layout is as follows:

```
rsp: Heap execute struct address
rsp + 0x08: Argument 1
rsp + 0x10: Argument 2
rsp + 0x18: Argument 3
...
```



Code within heap executes has some oddities. For instance, all external function calls must be offset with the "call offset" value placed on the heap. For example, to call `GetProcAddress`, letting `rdx` contain the call offset:

```
lea rax, GetProcAddress
call [rdx+rax]
```



For ease of reading, the instruction `lea rdx, [rip - 7]` at the beginning of the heap-executed code may be replaced with `xor rdx, rdx` to eliminate call offsets in decompilation.

Additionally, calls to `NtQuerySystemInformation` are replaced with `syscall` instructions. The register setup prior to `syscall` is as follows:

```
rax = syscall number from heap
r10d = 0xB9
rdx = address of arguments
r8d = size of arguments
r9 = 0
```



Prior to returning, a `NtQuerySystemInformation` system call is done with `rdx`, `r8`, and `r9` set to 0. This resets the stack frame before returning to prevent a crash.

## Kernel-Level Decryption

When `NtQuerySystemInformation` is called with a `SYSTEM_INFORMATION_CLASS` value of `0xB9`, the call proceeds to `WbDispatchOperation`. From here, the call proceeds to a specific function depending on the `operation` value provided in the arguments struct. The possible operations are summarized in the table below.

Operation Number	Operation
0	No-op
1	Decrypt encryption segment
2	Re-encrypt encryption segment
3	Heap execute
4	Heap execute return
5	Unused

Operation Number	Operation
6	Unused
7	Remove process
8	Unload module

All routines ensure that code decryption operations are done on read-only executable pages belonging to an binary signed as a "Windows System Component", presumably as a form of [exploit mitigation](#). The intrepid may easily bypass this restriction by simply mapping such a binary in their own code, as no other integrity checks are done by the kernel.

Code decryption is done within the kernel, and the round functions are executed by the function `WarbirdCrypto::CCipherFeistel64::CallRoundFunction`, which is called by a feistel decryption routine whose name is not available in public symbols. This routine has arguments as follows:

```
void FeistelDecrypt(
    FEISTEL_ROUND_DATA* rounds,
    BYTE* input,
    BYTE* output,
    ULONG length,
    UINT64 key,
    ULONG iv,
    BYTE* checksum
);
```



Knowing the address of this routine, a reverse-engineer can trivially use the `ntoskrnl.exe` binary to decrypt Warbird-encrypted binaries. Implementation of this method is left as an exercise for the reader.