Interestingly enough, the default compile settings in VS2017 don’t enable ASLR. The main function is the same address every time.

Do To:

* What is a stack cookie/canary? - [Corelan - Bypassing Stack Cookies, SafeSeh, SEHOP, HW DEP and ASLR](https://www.corelan.be/index.php/2009/09/21/exploit-writing-tutorial-part-6-bypassing-stack-cookies-safeseh-hw-dep-and-aslr/)

They are placed on the stack right before a buffer is allocated, that way if it overwrites, it will trigger an exception and exit securely. Stack canaries work by modifying every function's prologue and epilogue regions to place and check a value on the stack respectively. As such, if a stack buffer is overwritten during a memory copy operation, the error is noticed before execution returns from the copy function. When this happens, an exception is raised, which is passed back up the exception handler hierarchy until it finally hits the OS's default exception handler. If you can overwrite an existing exception handler structure in the stack, you can make it point to your own code. This is a Structured Exception Handling (SEH) exploit, and it allows you to completely skip the canary check.

* How can you bypass DEP? - [Corelan - Chaining DEP with ROP](https://www.corelan.be/index.php/2010/06/16/exploit-writing-tutorial-part-10-chaining-dep-with-rop-the-rubikstm-cube/)

DEP and NX essentially mark important structures in memory as non-executable, and force hardware-level exceptions if you try to execute those memory regions. This makes normal stack buffer overflows where you set eip to esp+offset and immediately run your shellcode impossible, because the stack is non-executable. Bypassing DEP and NX requires a cool trick called Return-Oriented Programming.

ROP essentially involves finding existing snippets of code from the program (called gadgets) and jumping to them, such that you produce a desired outcome. Since the code is part of legitimate executable memory, DEP and NX don't matter. These gadgets are chained together via the stack, which contains your exploit payload. Each entry in the stack corresponds to the address of the next ROP gadget. Each gadget is in the form of instr1; instr2; instr3; ... instrN; ret, so that the ret will jump to the next address on the stack after executing the instructions, thus chaining the gadgets together. Often additional values have to be placed on the stack in order to successfully complete a chain, due to instructions that would otherwise get in the way.

The trick is to chain these ROPs together in order to call a memory protection function such as VirtualProtect, which is then used to make the stack executable, so your shellcode can run, via an jmp esp or equivalent gadget. Tools like mona.py can be used to generate these ROP gadget chains, or find ROP gadgets in general.

* How can we bypass ASLR? - [ASLR/DEP bypass whitepaper](http://www.exploit-db.com/wp-content/themes/exploit/docs/17914.pdf) (PDF)

Direct RET overwrite - Often processes with ASLR will still load non-ASLR modules, allowing you to just run your shellcode via a jmp esp.

Partial EIP overwrite - Only overwrite part of EIP, or use a reliable information disclosure in the stack to find what the real EIP should be, then use it to calculate your target. We still need a non-ASLR module for this though.

NOP spray - Create a big block of NOPs to increase chance of jump landing on legit memory. Difficult, but possible even when all modules are ASLR-enabled. Won't work if DEP is switched on though.

Bruteforce - If you can try an exploit with a vulnerability that doesn't make the program crash, you can bruteforce 256 different target addresses until it works.

* What is a format string vulnerability
* How can we use that to know the address for the system function in windows.h
* Exploit to bypass DEP

Next week

* Compile with ASLR
* Calculate offset with format string
* Perform buffer overflow