

The Return Ticket

Overwriting where a function returns, and why not every modern defense stops it.

Course & track	CYBR 358, Cyber Operations (Writing-Intensive)
Term	Fall 2026
Instructor	Dr. Ahmet Furkan Aydogan
Email	aydogana@uncw.edu
Assigned / Due	Assigned week of _____ ; due two weeks later
Deliverable	Lastname_Firstname_BOF.docx, plus payload.bin, solve script, evidence
Points	100 (plus optional bonus)

When a function is called, it leaves itself a note on the stack that says where to go back when it finishes. Think of it as a return ticket. The processor reads that ticket, in one instruction called `ret`, and travels to whatever address is written on it. This lab is about what happens when an attacker gets to write on the ticket. You will overflow a small buffer, scribble a new destination onto the return ticket, and send the program somewhere it was never meant to go: a function called `win()`. That technique is called `ret2win`.

Then comes the part the old labs skipped. You will turn on four modern defenses, one at a time, and watch which ones actually stop you. Two of them do not, and understanding why is the point. Part I explains the stack, the return ticket, and the defenses; Part II is the graded walkthrough where you take the wheel and then, at the end, fix the bug for good.

1. Background

Overview and learning objectives

By the end of this lab you will be able to:

- Describe how a function's local buffer, saved frame pointer, and saved return address sit on the stack.
- Measure the exact distance from a buffer to the saved return address, instead of guessing it.
- Redirect execution into a chosen function by overwriting the saved return address (`ret2win`).
- Turn on stack canaries, NX, PIE with ASLR, and FORTIFY, and explain which stop the attack and which do not.
- Repair the vulnerability at its root with a bounded read, and test the repair at its edges.
- Frame all of this in the language of a cyber operation and of information assurance.

This lab sits in the exploitation, or action, phase of a cyber operation (CYBR 358 SLO 2 and SLO 3): gaining control of a target's execution. Its second half is pure information assurance. The same overflow, run against a hardened build, is where you see what each defense buys and what it costs. The written analysis you submit is the Writing-Intensive component.

Key terms

- Stack.** A region of memory a program uses for the local variables of the function it is currently running, stacked like cafeteria trays: the last one put down is the first taken back up.

- **Buffer.** A fixed block of bytes set aside to hold input, here a 64-byte array. It has an edge, and nothing in the vulnerable code checks whether you write past it.
- **Saved return address.** The return ticket. Just before a function runs, the address to resume at is saved on the stack, a little past the buffer. `ret` loads it back into the instruction pointer.
- **Offset (D).** The distance in bytes from the start of your input to the saved return address. You measure it. It is not simply the buffer size plus eight; the compiler decides the real layout.
- **ret2win.** Overwriting the saved return address with the address of a function you want to run (here `win()`), so the program returns into it.
- **Little-endian.** On x86-64 an eight-byte address is written least significant byte first. The tool `p64()` does this ordering for you.
- **Stack canary.** A secret value the compiler places between the buffer and the return ticket, like a thin tamper thread. If an overflow crosses it, the value changes, and the program aborts before it ever reads the ticket.
- **NX (non-executable stack).** A rule that memory holding data may not be run as code. It stops payloads that place instructions on the stack. Keep it in mind for later.
- **PIE and ASLR.** A Position-Independent Executable can be loaded at a different base address every run; Address Space Layout Randomization is what shuffles those addresses. Together they move the code each time the program starts.
- **FORTIFY.** A compiler and library feature that, for certain calls, knows the size of the destination buffer and refuses a copy that would overflow it.

The stack, from two chairs

Picture the vulnerable function's frame from low address to high: first the 64-byte buffer, then the saved frame pointer, then the saved return address (the ticket), then the caller's frame above that. Your input flows into the buffer from its start and grows toward higher addresses, straight at the ticket.

The mechanism on the defender's side. The saved return address is not a bug. It is how any function returns to its caller. The program is doing exactly what it was built to do: read the ticket, go there.

Now flip it. The same mechanism on the attacker's side. If your input can reach the ticket and you can write past the buffer's edge, then you choose where the function goes when it returns. You are not breaking the return; you are steering it. The one question the attacker asks is: how many bytes from the start of my input until I am writing on the ticket? That number is D.

Key idea. A buffer overflow here is not mainly about the crash. The crash is a symptom. The prize is the return address: control of where the program runs next.

Where this is used for real

The stack buffer overflow is one of the oldest control-flow attacks. The Morris Worm of 1988, one of the first programs to spread across the early internet, used a buffer overflow in a Unix service among its methods. Decades of overflow bugs followed, which is exactly why the defenses in Part II exist: stack canaries, non-executable memory, and address randomization were all added to make this attack harder. Studying the attack and its defenses together is how you learn to detect and defend against it. The technique is neutral; the target and the authorization are what make it right or wrong.

2. The assignment

Where your answers go. Do NOT hand in a blank document. Open the file “Buffer Overflow - Student Worksheet.docx” that came with this lab. Every step and question below has a matching answer line or screenshot box in that worksheet. Record every value, paste every screenshot, and write every explanation there. The completed worksheet is what you submit.

Step 0. Set up and check the environment

The lab ships as one Docker image, identical for everyone, so you install nothing but Docker. Build it, run it, and run the self-check FIRST. The check answers one question you will ask yourself many times over the next two weeks: is my environment broken, or is my measurement wrong?

Command reference

```
docker build -t cybr358-bof lab/
docker run --rm -it cybr358-bof
# inside the container:
/lab/lab-doctor.sh
```

What you should see (your numbers will differ)

```
CYBR 358 Buffer Overflow -- environment check
[ok] CPU/emulation reports x86_64
[ok] gcc present
...
[ok] gdb can start the program
Summary: 6 ok, 0 failed.
You are ready. Start with Part II of the handout.
```

If a line fails, fix it before going on; the Troubleshooting section at the end explains each one. On an Apple Silicon Mac the debugger may not run under emulation. That is a known platform issue, not your mistake; use the x86-64 path your instructor documents.

Step 1. Read the source and find the bug

Open `lab/src/vuln.c`. The whole vulnerability is two facts sitting next to each other: a 64-byte buffer, and a `read()` that copies up to 512 bytes into it. Nothing checks the count against the size. Everything past the 64th byte overflows.

Send input from a file, never by typing it. The program reads raw bytes, so your payload will contain bytes that a keyboard cannot type (including zero bytes and newlines), and only a file preserves them.

Step 2. Crash it on purpose

Feed it more than it can hold and watch it fall over. This is the symptom, not the prize.

Command reference

```
python3 -c 'import sys; sys.stdout.buffer.write(b"A"*200)' > crash.bin
gdb -q /lab/src/bof_baseline
(gdb) run < crash.bin
```

What you should see (your numbers will differ)

```
Program received signal SIGSEGV, Segmentation fault.
```

A segfault means the program tried to go to an address that was not valid. You overwrote the return ticket with `AAAAAAAA` and `ret` tried to travel there. Good. Now make it exact.

Step 3. Measure the offset D

Do not guess and do not count by hand. Use a cyclic (De Bruijn) pattern: it never repeats a four-byte window, so whichever four bytes land on the return ticket tell you the offset directly. Hints 1 to 3 in `lab/hints/` walk this in detail; open them freely, there is no penalty.

Command reference

```
cyclic 200 > pat.bin
gdb -q /lab/src/bof_baseline
(gdb) run < pat.bin
(gdb) info registers rsp rip
(gdb) x/gx $rsp
# take the value sitting on the return slot, then:
cyclic -l 0x6161616c
```

The number `cyclic -l` prints is D. Before you build the full payload, you may confirm the number at the offset checkpoint (an optional self-help endpoint your instructor provides). It only checks the number; it does not prove your measurement or run your program, and it is worth no points either way.

Step 4. Find the address of `win()`

The target is a function named `win()` that prints `LOCAL_WIN` and exits. The baseline binary is non-PIE, so `win()` lives at a fixed address you can read straight from the binary.

Command reference

```
objdump -d /lab/src/bof_baseline | grep '<win>:'
# or:
nm /lab/src/bof_baseline | grep 'win$'
```

Step 5. Take the wheel (ret2win)

Your payload is exactly D bytes of padding, then the eight-byte address of `win()`, and nothing after it. The scaffold `lab/solve.example.py` has two blanks to fill: your measured D and the address. Run it. It writes `payload.bin` and fires it at the program.

Command reference

```
# fill in OFFSET and WIN_ADDR in solve.example.py, then:
python3 solve.example.py
```

What you should see (your numbers will differ)

```
[+] Starting local process '/lab/src/bof_baseline'
LOCAL_WIN
```

When `LOCAL_WIN` appears, you did not crash the program. You steered it. The return ticket now carries an address you chose, and the processor obeyed.

Money shot. Paste a screenshot that shows your command producing `LOCAL_WIN` on `bof_baseline`, in the same view as your `payload.bin` being created. Without this screenshot the `ret2win` step cannot be graded.

[Screenshot 1 goes here]

Step 6. Turn on the defenses, one at a time

Now the information-assurance half. The Makefile builds five more binaries, each an INDEPENDENT build from the same source, so you test one defense at a time and never stack them. Build them, run your attack against each, and record what happens. Do not just write “it crashed.” Show the mechanism.

Command reference

```
make -C /lab/src all
```

Build	What you should observe	The mechanism to explain
baseline	ret2win succeeds, even though NX is ON.	NX forbids running code on the stack. ret2win runs NO stack code; it reuses an existing function. So NX does not apply.
canary	The program aborts at return: stack smashing detected.	A secret value between the buffer and the ticket changed when you crossed it, and the check fired before <code>ret</code> .
PIE + ASLR	Your fixed-address exploit now fails.	<code>win()</code> is at a different address every run. The single number you hard-coded is no longer where <code>win()</code> lives.
FORTIFY on	The program aborts on the read: buffer overflow detected.	At <code>-O2</code> the library knows the buffer is 64 bytes and refuses a 512-byte read into it.

For each build, fill in the worksheet’s expectation, observation, and mechanism. Two of these (NX and ASLR by itself against this non-PIE baseline) are chosen so you can feel the difference between a defense that changes the outcome and a property that does not. When you compare the FORTIFY builds, compare `fortify_on` against `fortify_off` (both at `-O2`), not against the baseline, so only the one variable changes.

Watch out. After any rebuild, re-measure. An old payload failing does not by itself prove a defense worked; the layout may simply have shifted. For the FORTIFY-off control, confirm your ret2win still lands before you claim FORTIFY-on stopped it.

Step 7. Fix it, and test the fix

Open `lab/src/vuln_fixed.c`. The read is now bounded to the buffer, the result is always terminated, and any extra input is drained instead of written. Build it and prove it holds.

Command reference

```
make -C /lab/src bof_fixed
printf 'alice\n' | /lab/src/bof_fixed # normal input works
python3 -c 'import sys;sys.stdout.buffer.write(b"A"*500)' | /lab/src/bof_fixed # over-long is
↪ safe
```

A fix is not “it did not crash.” Test it at the edges: empty input, an input exactly at the buffer size, an over-long input, and a normal name. Explain in the worksheet WHY the bounded read closes the hole, not just that it does.

3. Analysis questions

Answer every question in the Word worksheet (record every answer there).

- Q1. The measurement.** What is your measured D, and what address did `win()` have in the baseline? Show the exact command output you read each value from.
- Q2. Identify.** Name the three things between the start of your buffer and where `ret` sends the program, in order. Which one did your padding have to cross to reach the ticket?
- Q3. In your own words.** Explain `ret2win` to a first-year student, using an analogy that is NOT the return ticket from this handout.
- Q4. The limits of the evidence.** Your fixed-address exploit failed against the PIE build. Does that prove address randomization stops buffer overflows in general, or only that it stops THIS technique, which relied on a known address? What would an attacker still need?
- Q5. Trade-off and defender's move.** NX was on the whole time and never stopped you. Explain why, and then, from the defender's chair, name two defenses from this lab you would ship together and what each one costs.

4. Deliverable, grading, and ethics

What to submit

The completed worksheet (Lastname_Firstname_BOF.docx), your `payload.bin`, your solve script, and your evidence (the debugger dumps and run transcripts the worksheet asks for). Evidence should be copyable text where possible, not only screenshots.

Grading (100 points)

Criterion	Points
Measurement and explanation (how you found D and the address, not just the values)	30
Local <code>ret2win</code> and reproducibility (<code>LOCAL_WIN</code> , <code>payload.bin</code> , byte-level account)	30
Mitigation experiments (expectation, observation, mechanism for each build)	20
Safe fix and boundary tests	20
Total	100

The offset checkpoint and the payload validator are self-help only and are worth no points. Optional bonus (up to 10, on your own machine, documented in your write-up): place your own shellcode on the stack and run it with NX turned off, or reach a libc function with a short ROP chain. Bonus never substitutes for the safe fix or the core explanation.

Honesty

A correctly reported “the canary stopped me” or “PIE broke my exploit” earns full marks; a fabricated `LOCAL_WIN` or an invented offset earns zero. If you got partway, report exactly how far and what you saw. A precise negative is worth more than a fake success.

Academic integrity, scope, and ethics

This is individual work. The program is deliberately vulnerable and is meant for your OWN lab machine or the provided container, on synthetic input with no real person or system involved. Running an exploit against software you do not own or are not authorized to test is a crime, not a

lab. The bright line is authorization and ownership: this is legal on the target you own or are given, and never on anyone else's. It is taught so you can detect and defend against it, which is the entire second half of this lab. AI may help you understand the tools, but measure every value and run every command yourself; a number you cannot reproduce in the debugger is a number you did not earn. The technique is neutral; the target and the authorization are what make it right or wrong.

5. Troubleshooting

- **It printed LOCAL_WIN once, then a later run crashes on the same payload.** You probably rebuilt with different flags. Re-measure D and the address for the exact binary you are attacking.
- **The address is right but it still segfaults inside win().** This lab already aligns the stack for you in `win_stub.S`, so you should not hit the classic alignment crash. If you do, confirm you are attacking `bof_baseline`, not a rebuilt variant.
- **My eight-byte address looks backwards.** It is little-endian. Least significant byte first. `p64()` already does this; do not reverse it yourself.
- **gdb will not start the program (Apple Silicon).** The image runs under emulation on ARM and the debugger can fail there. Use the x86-64 virtual machine or lab-machine path your instructor documents. This is a platform limitation, not your error.
- **The first time takes a while.** Measuring an offset and reading a debugger for the first time is slow, and that is normal. Nothing here is dangerous to your computer.