CS-412 Final Exam (Spring 2022)

May 31, 2022

SCIPER:_____

Last name:_____

First name:_____

1. This is a closed book exam. No extra material is allowed. If you have a question, raise your hand and wait for the proctor.
2. You have 75 minutes, and there are 75 points. Use the number of points as guidance on how much time to spend on each question.
3. Be sure to provide (print) your name. **Do this first so you don't forget! Please write or print legibly**. State all assumptions that you make above those stated as part of a question.
4. Write your answers directly in the dedicated boxes. If you need more space your answer is probably too long.
5. Leave your CAMIPRO card on the table so it can be checked.
6. With your signature below you certify that you solved these problems on your own, that you turn in your solution, and that there were no environmental or other factors that disturbed you during this exam or that diminished your performance.

Signature:_____

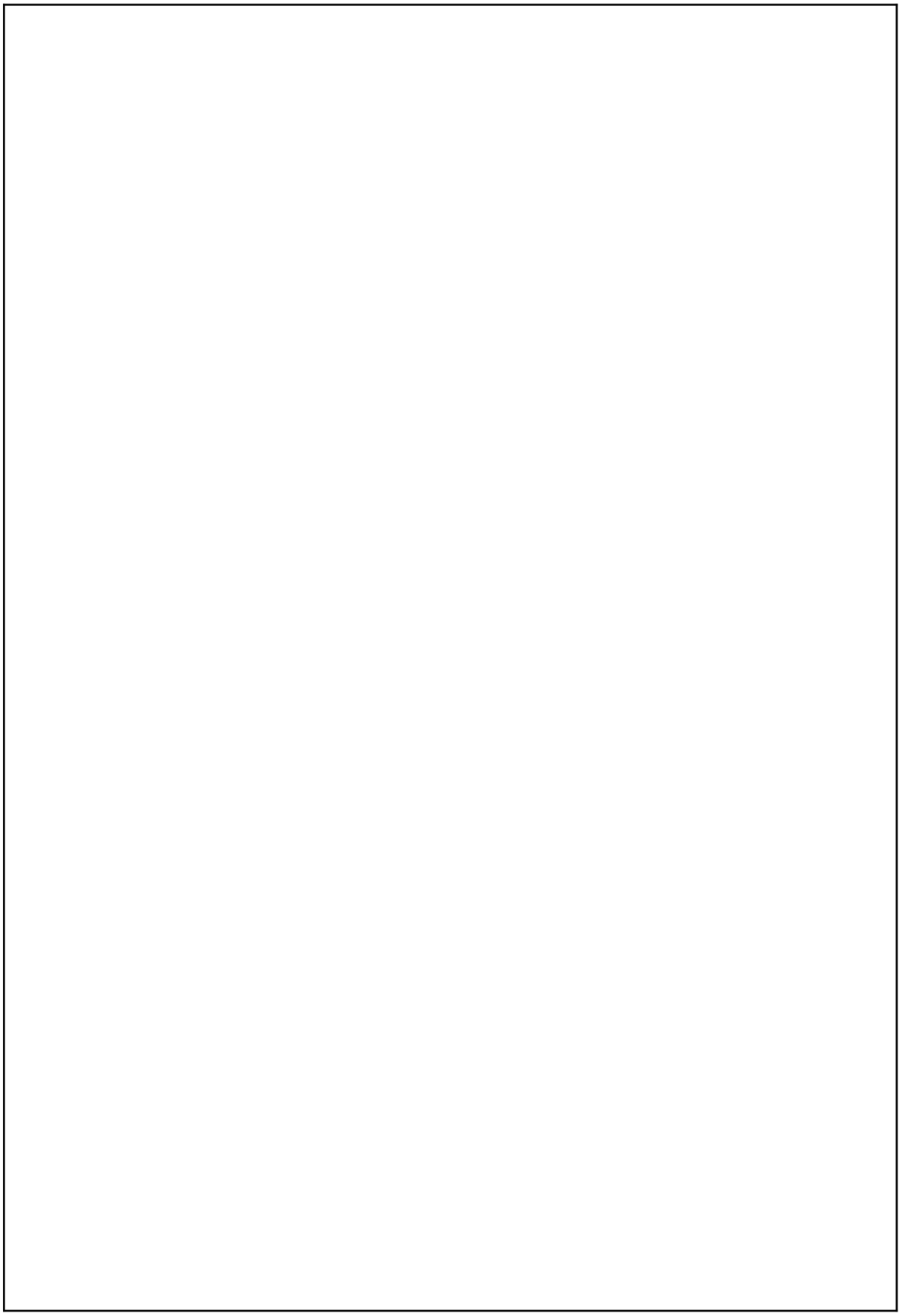
Question	**Points**
1	/20
2	/15
3	/20
4	/20
Total	

Q1) Containing Exploitation Challenge (20 points)

You're a TA for the SoftSec lab, and you need to set up the infrastructure for the CTF lab. Each challenge needs to run in its own separate environment, safe from possible exploitation from other vulnerable challenges. The scoreboard service also needs to be separated from the challenges. The scoreboard service manages the database with the current status of the ctf (e.g., score per user, login information, or solved challenges) and serves the user-accessible website. Users can access and look at challenges, scores, reset their passwords, and so on.

Describe how you would set up all the above services to compartmentalize and avoid compromise. You are free to split services into multiple smaller ones. If your setup involves multiple machines, draw a graph to represent their interactions. If your setup involves a containerization mechanism (e.g., Docker, or virtual machines) state which one you picked and for what reason.

Hint: *multiple users might want to try the same challenge at the same time. Make sure each one has their own unique experience, without interference from other users.*



Q2) HW-assisted Control-Flow Integrity (15 points)

Intel CET (Control-Flow Enforcement Technology) provides, among other elements, a new instruction for forward-edge control-flow integrity (`endbr64` and `endbr32` for 64 and 32 bit systems respectively). This instruction must be included right before all functions which are targets for indirect calls or indirect jumps. For example, any target of an indirect control-flow edge must necessarily start with the "`endbr64`" instruction. If an indirect call or branch lands on a different instruction, the CPU will trigger a fault and stop execution.
Also, we encode the "`endbr64`" in such a way that older CPUs without this feature will just interpret the "`endbr64`" as a NOP instruction, ensuring compatibility with older processors.

1. What are the **performance disadvantages** in the introduction of Intel CET `endbr` instructions for protected applications (think about the binary, the operating system support, the execution of the program, and the performance of the overall system)?

2. What kind of exploitation techniques does Intel CET `endbr` instruction **mitigate**?

3. Why can `endbr` not protect against Return-Oriented Programming?

4. Does `endbr` **really** stop jumping to any non-valid indirect branch target? Can you think of a way how an attacker could still jump in the middle of a function?
   (Hint: x86 is a weird architecture. For example, think about how there are many more ROP gadgets than the number of compiler-generated ret instructions… why?)

5. What is the target set of this particular implementation of CFI? Why is it considered "easy" to bypass such an implementation of CFI?

Q3) Mystery Mitigations (20 points)

The company you work for has messed up, they were transferring their whole infrastructure from one server to another, and the process has corrupted a couple of files, making them illegible. One of the corrupted files contained the compilation toolchain details, and your task is to **identify the applied mitigations** back from the object files **and tell what their concrete purpose is**. There were two mitigations in the toolchain, so you get the codebase's simplest function source code and the 4 resulting compiled object files (without mitigations, with mitigation 1, with mitigation 2, and with both mitigations).

You spot that, when applying these unknown concrete mitigations, there are major changes in the functions. The following assembly snippets (Intel syntax) show the changes in the compiled function (sections that have the same purpose between versions are highlighted in the same color).

Aside from identifying the mitigations, **point out all the other existing mitigations you know that offer similar safety guarantees**. Your ability to read x86 assembly may have gotten a bit rusty, luckily you find a cheatsheet you wrote back when you took CS-412:

x86 Cheat Sheet (Intel syntax)

[a] denotes the memory contents pointed by address **a**, **a** is an arithmetic operation of registers and scalars (e.g., rax+rbx*8-0x10)

The FS segment is commonly used to address Thread Local Storage (TLS). When referencing a TLS slot, we do so by **fs:offset**

All memory accesses are assumed to use **QWORD PTR**, which is for 4 words (64 bits) memory access granularity

<VAR> denotes a program variable. If a value doesn't match with the C function variables, it is given by the compiler (e.g., *<M>*)

l: label where to jump/call

call a	pc = a **push** rsp	**push** a	rsp -= 8 [rsp] = a	**pop** a	a = [rsp] rsp += 8
mov a, b	a = b	**lea** a, [b]	a = b	**add** a, b	a = a + b
cmp a, b	*EQ* = a == b	**jne** a	pc = a if not *EQ*	**sub** a, b	a = a - b

C function snippet

```
uint64_t A[L];
void dumb(size_t n, uint64_t e) {
  uint64_t array[L];
  memcpy(array, A, L*sizeof(uint64_t));
  array[n] = e;
  memcpy(A, array, L*sizeof(uint64_t));
  return;
}
```

No mitigation	Mitigation 1	Mitigation 2	Mitigation 1 & 2
push r15	push r15	push rbp	push rbp
push r14	push r14	push r15	push r15
push r12	push r12	push r14	push r14
push rbx	push rbx	push r13	push r13
		push r12	push r12
		push rbx	push rbx
		push rax	push rax
sub rsp,<L>+8	sub rsp,<L>+8		
mov r14,rsi	mov r14,rsi	mov r14,rsi	mov r14,rsi
mov rbx,rdi	mov rbx,rdi	mov r15,rdi	mov r15,rdi
		mov r13,[<M>]	mov rax, [<M>]
		mov rbp, fs:[r13]	mov rbp, fs:[rax]
		lea rbx,[rbp-<L>]	lea rbx,[rbp-<L>+0x10]
		mov fs:[r13],rbx	mov fs:[rax],rbx
	mov rax, fs:0x28		mov r13,fs:0x28
	mov [rsp+<L>],rax		mov [rbp-0x8],r13
lea r15,[<A>]	lea r15,[<A>]	lea r12,[<A>]	lea r12,[<A>]
mov r12,rsp	mov r12,rsp		
mov edx,<L>	mov edx,<L>	mov edx,<L>	mov edx,<L>
mov rdi,r12	mov rdi,r12	mov rdi,rbx	mov rdi,rbx
mov rsi,r15	mov rsi,r15	mov rsi,r12	mov rsi,r12
call <memcpy@plt>	call <memcpy@plt>	call <memcpy@plt>	call <memcpy@plt>
mov [rsp+rbx*8],r14	mov [rsp+rbx*8],r14	mov [rbp+r15*8-<L>],r14	mov [rbp+r15*8-<L>+0x10],r14
mov edx,<L>	mov edx,<L>	mov edx,<L>	mov edx,<L>
mov rdi,r15	mov rdi,r15	mov rdi,r12	mov rdi,r12
mov rsi,r12	mov rsi,r12	mov rsi,rbx	mov rsi,rbx
call <memcpy@plt>	call <memcpy@plt>	call <memcpy@plt>	call <memcpy@plt>
	mov rax,fs:0x28		
	cmp rax,[rsp+<L>]		cmp r13,[rbp-0x8]
	jne f		jne f
			mov rax,[<M>]
		mov fs:[r13],rbp	mov fs:[rax],rbp
add rsp,<L>+8	add rsp,<L>+8	add rsp,0x8	add rsp,0x8
pop rbx	pop rbx	pop rbx	pop rbx
pop r12	pop r12	pop r12	pop r12
pop r14	pop r14	pop r13	pop r13
pop r15	pop r15	pop r14	pop r14
		pop r15	pop r15
		pop rbp	pop rbp
ret	ret	ret	ret
	f: call <fail>		f: call <fail>

1. Name and explanation of **Mitigation 1**:

2. Name and explanation of **Mitigation 2**:

3. Comparison with other mitigations that offer similar safety guarantees

Q4) Better Crash Debugging (20 points)

We aim to design a human-in-the-loop crash debugger that facilitates identifying the root cause of a crash. The debugger would require the developer's intervention whenever an assumption needs to be made or an ambiguity resolved. Starting from a crashing test case, the debugger should help the developer trace the execution back to the root cause. The root cause of a bug is the first point in the code where an assumption is violated or unexpected behavior is observed.

1. How can the debugger leverage concolic execution to identify the culprit? What is the role of the developer in guiding concolic execution?

2. Is concolic execution sufficient to identify the root cause of **ANY** bug? Justify.

3. The developer is now interested in determining whether a certain assumption in their code can be broken by some input. The assumption can only be evaluated inside a function buried deep in their code. To assist in such scenarios, the debugger should also support backward-slicing. Backward-slicing is the process of symbolically constructing partial paths (slices) that reach or affect a certain location or variable in the program. How can backward-slicing be helpful in this case?

4. What data structure must the debugger build from the program to identify and explore the different paths that could reach the function in question?

5. In practice, results obtained from the described backward-slicing method may be unsound. What are the reasons and sources for such unsoundness? Justify.

6. What is the main challenge for ensuring that this analysis is complete? Give an example from your experience where you encountered incompleteness of symbolic execution.