# CS 426: Computer Security
# Fall 2025
# Homework 5: Sandboxing and Isolation

**Due: Oct 7th, 2025 @ 11:59 PM**

## 1 Overview

The goal of this assignment is for you to explore sandboxing as a security mechanism by comparing the behavior of vulnerable C library code in two environments: with and without sandboxing protection. Sandboxing isolates potentially dangerous code execution to prevent it from compromising the broader system.

## 2 Environment Setup

**For this assignment, use a Linux x86 environment.** You may work on your own laptop or on Purdue CS servers. Do not use the Kali VM – it is 32-bit and some required tools cannot be installed there.

## 3 Assignment Setup

For this assignment, you will sandbox unsafe code using two tools. First, install both tools. Then, download the starter code and update the Makefile with their paths. **If you are using Purdue data server for this assignment, both tools are already installed under `/homes/cs426/emsdk` and `/homes/cs426/wabt`, so you can skip steps 1 and 2.**

1. **The Emscripten toolchain.** Emscripten is a complete compiler toolchain to WebAssembly with a special focus on speed, size, and the Web platform. We will use it to compile unsafe C code into sandboxed WebAssembly.

   ```
   git clone https://github.com/emscripten-core/emsdk.git
   cd emsdk
   ./emsdk install latest
   ./emsdk activate latest
   ```

2. **The WebAssembly Binary Toolkit.** WABT (we pronounce it ”wabbit”) is a suite of tools for WebAssembly, including *wasm2c* which converts a WebAssembly binary file to a C source and header. We will use it to translate the compiled WebAssembly modules into portable C code so that we can easily use them.

```
git clone --recursive https://github.com/WebAssembly/wabt
cd wabt
git submodule update --init
mkdir build
cd build
cmake ..
cmake --build .
```

3. **HW5 Code.** The code for this assignment is available on both BrightSpace and Purdue Box. Use the following commands to download it.

```
wget -O hw5.tar.gz \
    https://app.box.com/shared/static/kw1p0md8ag3b858hmgm3i3c8htlnx6sf.gz
tar zxvf hw5.tar.gz
cd hw5
```

   Then, update the paths in the Makefile to point to the tools you installed.

```
~/hw5$ cat Makefile
EMSDK_DIR=/path/to/emsdk/      # FIXME: fill in the absolute paths to emsdk/
WABT_DIR=/path/to/wabt/        # FIXME: fill in the absolute paths to wabt/
```

Now you can start running each binary and observe their different behaviors. Here are some resources that will help you in this assignment:

**WebAssembly:** WebAssembly (abbreviated Wasm) is a safe, portable, low-level code format designed for efficient execution and compact representation. To learn more about its design goals and runtime structures, you can refer to its documentation.

**Fine Grain Isolation and Sandboxing:** Major applications including web browsers rely on many third-party libraries. These libraries often contain security vulnerabilities that can compromise the main application. One way to mitigate this threat is to sandbox the libraries to prevent the main application from getting compromised. The Retrofitting Fine Grain Isolation in the Firefox Renderer paper discusses the primitives of sandboxing and develops a framework based on WebAssembly called RLBox for this purpose.

**Note:** Please note that you don't need to go through all the documention and provided resources to complete this assignment. Although, reading them is strongly recommended to get familiar with how sandboxing in WebAssemly works.

## 4 Sandboxing Unsafe Library

Software engineer Milad is writing an application in C. He found a library (`lib.c`, `lib.h`) on GitHub that does some useful work. In detail, the library provides the following useful interfaces:

- `get_score(char *username)` – returns the letter grade for `username`.
- `get_hash(char *s)` – returns a numeric hash of the input string `s`.
- `set_callback(callback_t cb)` – register `cb` as the callback used by `get_factorial`.
- `get_factorial(long n, char *result_msg)` – computes $n!$ and invokes the registered callback with `result_msg` and the factorial result.

Unfortunately, the library seems to be poorly written and contains many vulnerabilities.

### 4.1 app-v1: A Naive Approach

Despite of the security concerns, Milad implemented his application in `app-v1.c`, using the unsafe library. The program takes in 3 cmdline arguments and calls into the library interfaces with the arguments. Just like all the other programs, Milad's simple program also holds some secrets to protect.

**Task 1 (15 points)** With all the knowledge you learned in CS426, you want to convince Milad that his program is at risk! Convince him by following these steps:

1. Which functions are vulnerable and why? What harm can be done? Identify the vulnerabilities and explain exactly how they are vulnerabilities.
2. How can you exploit the identified vulnerabilities? Describe your attack strategies.
3. You are encouraged to implement your attacks as Python scripts and include proof (screenshots) of successful exploitation. Otherwise, give concrete details on why such attacks are practical (exact payload length, target address to overwrite, overwrite value, etc.).

Each identified vulnerability is worth 5 points. To receive full points for this task, you must identify 3 vulnerabilities and explain them by following the steps above.

**Notes.** To compile the program and perform attacks, run the following commands:

```
make app-v1                              # compile app-v1
make test-v1                             # test app-v1 with attack scripts sol*.py
setarch $(uname -m) -R ./app-v1 <NAME> <STR_TO_HASH> <MSG_ON_FINISH>       # test app-v1 manually
```

Notice that you must disable ASLR if you are hijecking the control flow to one of the `print_secret` functions. This can be done with the `setarch` command above.

### 4.2 app-v2: Sandboxing with WebAssembly

Thanks to your efforts, Milad realized the risk of directly calling the unsafe library — an attacker can exploit bugs in the library to trick the application or reveal secrets. Milad sandboxed the unsafe library with WebAssembly using the following steps:

1. **Compile to WebAssembly.** Compile the unsafe library from C (`lib.c`, `lib.h`) to WebAssembly (`lib.wasm`) with Emscripten.

   Compiling the code to WebAssembly places it in a sandbox, blocking direct access to the outside environment. The sandboxed code cannot read or write host memory, and it has no capabilities beyond pure computation. To perform external actions (e.g., file I/O or getting the time), the host must explicitly provide safe functions for the module to call.

2. **Convert Wasm back to C.** Produce an equivalent C source and header (`lib.wasm.c` and `lib.wasm.h`) from the WebAssembly (`lib.wasm`), using `wasm2c` tool in the WABT toolchain.

   Compiling the WebAssembly into C code allows it to be compiled into native X86 binary later, while preserving all the wasm semantics including sandboxing. Otherwise, a wasm VM is needed to run the WebAssembly, which involves more code changes in `app-v2.c` and performance overhead. Further, converting wasm back to C code allows one to read, verify, and modify the generated code easily.

3. **Integrate the sandbox in the application (app-v2).** Replace direct unsafe library calls with calls into the `wasm2c`-generated API:

   - The application first instantiates the WebAssembly module so that the unsafe code will run inside its own linear memory.
   - Before calling library functions, input strings from the host are copied into sandbox memory using the module's allocator and direct memory writes.
   - Sandbox functions are then invoked through the generated wrappers, which take sandbox memory offsets instead of raw host pointers.
   - Callbacks are registered by inserting function references into the sandbox's funcref table, and they are passed inside the sandbox as table indexes.
   - Finally, sandbox allocations and the instance itself are freed.

To learn more about how sandboxing works, review relevant code including `app-v2.c`, `wabt/wasm2c/wasm-rt.h`, `lib.wasm.h`, and `lib.wasm.c`. You can also find resources online, e.g. Wasm Security, Wasm2c README, WasmBoxC: Simple, Easy, and Fast VM-less Sandboxing, etc.

**Task 2 (30 points)** Help Milad verify whether sandboxing removes each vulnerability you found in `app-v1`. For every vulnerability do the following:

1. State clearly: Is this vulnerability prevented by sandboxing? (Yes / No)

2. Explain why the previous attack no long works (or still works). Provide supporting evidence (e.g. exact payload length, target address to overwrite, overwrite value). If the vulnerability remains, you are encouraged to implement your attacks as Python scripts and include proofs (screenshots) of successful exploitation.

3. Explain the exact mechanism of wasm sandboxing that (fails to) prevent this vulnerability (e.g., sandbox cannot call syscalls directly; sandbox returns offsets not host pointers).

The analysis of each vulnerability is worth 10 points. To receive full points for this task, you must analyze 3 vulnerabilities and explain whether they are mitigated by sandboxing by following the steps above.

**Notes.** To compile the program and test the attacks, run the following commands:

```
make app-v2                                # compile app-v2
make test-v2                               # test app-v2 with attack scripts sol*.py
setarch $(uname -m) -R ./app-v2 <NAME> <STR_TO_HASH> <MSG_ON_FINISH>       # test app-v2 manually
```

Same as before, you must disable ASLR if you are hijecking the control flow to one of the `print_secret` functions. This can be done with the `setarch` command above.

## 4.3 app-v3: Make it Safer

Thanks to your efforts, Milad realized that sandboxing cannot prevent all forms of attacks, including confused-deputy attacks. A confused-deputy attack occurs when a less-privileged component (sandboxed library) tricks a more-privileged one (host program) into misusing its authority. Therefore, Milad asks you to harden the application further.

**Task 3 (15 points)** Implement robust host-side validation in `app-v3.c`. Your job is to validate and safely copy any data received from the sandbox before the host uses it. The goal is that malformed or malicious sandbox outputs cannot crash the host or cause it to perform privileged actions or reveal secrets. You only need to modify `app-v3.c`; do not change `lib.c` or `lib.wasm.c`.

After you finish:

1. Attach a screenshot showing the fix in the updated program `app-v3`.
2. Briefly explain what your changes do and why they block the confused-deputy attack you identified in `app-v2`.
3. Is there any vulnerability remaining? If so, briefly explain why sandboxing and your fixes in `app-v3` cannot prevent it.

In this task, each question is worth 5 points.

**Notes.** To compile the program and test your fix, run the following commands:

```
make app-v3                                # compile app-v3
make test-v3                               # test app-v3 with attack scripts sol*.py
setarch $(uname -m) -R ./app-v3 <NAME> <STR_TO_HASH> <MSG_ON_FINISH>       # test app-v3 manually
```

Same as before, you must disable ASLR if you are hijecking the control flow to one of the `print_secret` functions. This can be done with the `setarch` command above.

## 5 Deliverable

### 5.1 What and How to Submit

For this assignment, you only need to submit a report on **Gradescope** under *Homework 5: Sandboxing(Report)*. Please assign pages to each attack on Gradescope.

| **File** | **Description** | **Submission** |
|---|---|---|
| `report.pdf` | Your write-up addressing all three tasks in Section 4. | **pdf** on Gradescope [Homework 5: Sandboxing (Report)] |

Table 1: List of required submission files

### 5.2 Homework Report Instruction

Your submission should include a report/write-up containing enough instructions and technical details to prove the **validity** and **reproducibility** of your work and demonstrate your understanding of each problem and your solution. Your document should contain answers to all questions in Section 4.

**Please be aware**: It is your responsibility to make sure your responses are clean and organized enough to be readable. Similar to previous homeworks, meaningless or frivolous responses such as "*because it is correct*", "*trivial*", and "*left as an exercise to the graders*" will be considered completely invalid.

On the last page of your report, include a **collaboration statement**. It can simply be a list of people you communicated with and the online resources you have used for this homework. If you have finished this assignment entirely on your own AND did not consult any external resources, you MUST indicate that on your collaboration statement. This is a part of your assignment. **Failure to provide one could result in a deduction of points**.

## 6 Grading and Policy

### 6.1 Grade Breakdown

This assignment has three tasks: the first is worth 15 points, the second 30 points, and the third 15 points. Altogether, the homework counts for 4% of your final grade.

### 6.2 Important Rules and Reminders

This assignment MUST be completed using only `gdb` and `python3`, along with necessary general-purpose tools such as `gcc` and `make`. DO NOT use other special-purpose exploit tools such as `ghidra`. No points will be given to a solution that utilizes such tools. For all tasks, DO NOT try to create a solution that depends on you manually setting environment variables.

You are more than welcome to use any external resources including the ones available online besides the ones provided to you already. However, the academic integrity policy still holds. **Do not ask someone else to do your work for you** or copy and paste the questions on websites to find answers. **Use your resources reasonably, and do not cross the line**. See the **Academic Integrity** section of the syllabus (course webpage) for more details.