Final Project

1 Requirements

The goal of this project is to become familiar with analyzing (statically and dynamically) software to detect security issues. Experiments should be conducted by trying out different approaches, and also providing some comparison of the pros and cons and effectiveness of various approaches. Please read the following guidance to choose the target software and security analysis methods.

- Targets: you can target any software you want. The only constraint is that it should be open source. You can choose some old outdated software, where it should be easier to find bugs, or some newer up-to-date software, where finding bugs may be harder but also more interesting. To keep things simple, choose some software that can take a single file as input on the command line. To make things rewarding, choose some software that takes a complex input format as input, e.g., software that handles graphical data. The software must be written in one of the following language:
    - C/C++
    - Solidity (Smart contract)
    - Rust
- Approaches: Perform experiments with at least three tools/systems. No matter they are static or dynamic analysis tools. You can try the tools/systems in **Section 5**. You are more welcome to try out other tools that you happen to know and are curious about. This project assignment is quite open-ended, so please plan to do over the next two months. We'll regularly review progress and share experiences in class, allowing us to adjust or refocus efforts as needed based on the results, or even wrap things up early if certain directions seem less fruitful. Optional things to try, depending on the experience with your specific target applications, include:
    - investigate (and fix?) any bugs you found;
    - check bugs found against known CVEs;
    - introduce bugs and see if tools can find these;
    - if you do not find bugs: test older releases;
    - try different settings of the tools, or different initial seeds;
    - or other creatives ideas of your own!

2 Registration

Fill in the following Google docs file. A group can have up to 4 members.

Registration link will be announced on the class's Teams channel.

3 Checkpoints

There will be a checkpoint. On those days, each group must present the work in progress. You must pass all the checkpoints before having the final presentation.
The schedule for checkpoints will be announced later.

4 Final submission and presentation.

You are required to submit the report, source code, and presentation slides. The submission link will be announced later. On the presentation day, all members of the group must attend the class and each member is recommended to present his/her own part.
Deadline: TBD

5 Approaches for software security

5.1 Solidity

5.1.1 Static analysis tools

- Slither
- Halmos
- Certora
- Mythril
- Kontrol
- hevm
- Oyente
- Osiris
- Maian
- TeEther
- Sereum
- ILF
- Vandal
- Madmax
- Ethir

- Smartcheck
- SaferSC
- RecChecker
- KEVM
- Eth-Isabelle
- SmartPulse
- Semgrep
- C4udit
- Cyfrin Aderyn

- Applied LLM (AI) for analyzing

5.1.2 Dynamic analysis tools

- ItyFuzz
- Echidna
- Scribble
- ContractFuzzer

5.2 C/C++

5.2.1 Dynamic analysis tools (Fuzzers)

More information and pointers about these fuzzers on tools page. In Brightspace there are discussion forums to exchange good and bad experiences on installing and using the tools. You can also have a look at this very gentle introduction to zuff, ASan and afl.

- Radamsa Radamsa is a file-based mutational fuzzer written in Owl-Lisp, a dialect of LISP. It takes a set of valid example files as input and then outputs an arbitrary number of invalid files by mutating them. It does not feed these files to the target software. You can feed these files manually to the target software or set up some automation for this.

  To install:

  ```
  > git clone https://gitlab.com/akihe/radamsa
  > cd radamsa
  > make
  > sudo make install # optional, you can also launch radamsa by giving the full path to bin/radamsa
  > radamsa --help
  ```

- zzuf zzuf is another mutational fuzzer, very similar in style to Radamsa. Indeed, it will be interesting to see if zuff and Radamsa produce very similar results. It is pretty old and it's not clear how actively it is being maintained. But it is simple, lightweight, and deterministic, so still worth a shot to try out. *To install*: On Linux, you can simply get it with sudo apt-get install zzuf

Alternatively, unpack the tar-ball from the git repo and run the standard sequence

```
> ./bootstrap
>  ./configure
>  make
>  sudo make install
```

- afl or afl ++ afl by Michael Zalewski is a 'smart' fuzzer that takes evolutionary approach to fuzzing: it mutates sample input files and then observes executions to see which mutations result in different code execution paths, to then prioritise these mutations over others. To observe the code execution path, the target software needs to be instrumented at compile time. (It is possible to use the tool if you do not have access to the source code by running the code in the QEMU emulator, but we're not going to use that option for this project.) The original afl is no longer maintained, though it should still work. There is a newer daughter project afl++ that might prove more stable. The QuickStartGuide.txt provides a quick intro. There is a daughter project afl++ which may be easier to install and run than the original afl. *To install*: The easiest way to install afl is with

  sudo apt-get install afl If that does not work on your Linux/Mac OS X machine, Peter Guttman's article Fuzzing Code with afl walks through the installation of both clang (as part of LLVM) and afl The instruction for afl++ are slightly differently. Unlike with the tools above, for afl you need to recompile your code before you can start fuzzing. This tutorial walks trough a simple example using afl. Peter Guttman's article Fuzzing Code with afl walks through the use of afl in combination with ASan. There is also an afl tutorial at fuzzing-project.org, which also discusses the use of afl with ASan. > **NB Windows users**: Originally afl only ran on Linux and Mac OS X, as it uses *nix features. You can run afl in a VirtualBox VM, with a performance hit of course, but using a memory-instrumentation tool like ASan is then not an option. There is now also a Windows fork of afl. If you have any Linux or Mac OS X users in the group, obvious strategy would be to use afl on these systems and not to try this Windows fork. Or, if you could try winafl for comparison; they had to make some tweaks for winafl, not sure how these will affect results. There is also a recent - still experimental - port of ASan for Windows. By all means give it a try, if you like a challenge, but you'd probably want to ditch it if you hit any snags. Please use the Brightspace forum to report good or bad experiences, so that other groups can benefit from this.

- CERT BFF Playing around with CERT BFF was not a success last year, so better give it a miss. BFF started off as a fuzzing framework built around zzuf, but has since evolved to use a different mutational fuzzer under the hood. BFF has grown out of the merger of two fuzzing tools, BFF for Linux/OSX and FOE for Windows, and includes several features to help not just in finding flaws but also analysing them: it tries to automate some of the configuration of the fuzzer, can try to to reduce fuzzed examples that cause crashes to a minimal size, and collects debugging information about crashes to help in analysis. Not sure if all this complexity makes the tool easier to use for this project than a bare bones tool like zzuf or harder… *To install*: from the BFF download page you can get OS X and Windows installers and a UbuFuzz virtual machine. Or you can grab the source code from the github, but that is probably not for the faint-hearted.

- HonggFuzz HonggFuzz is another modern fuzzer. It has been reported to outperform afl(++) in many cases: it will be interesting to see if it will also be the case for the target you are looking at.

5.2.2 Instrumentation tools

Below some instrumentation tools to try out. These should help to spot more errors when fuzzing. At least try out ASan, which will probably give you the best improvements for the effort and overhead. If you have trouble with ASan, you could use valgrind's Memcheck as a fallback.

- AddressSanitizer (ASan) ASan adds memory safety checks to C(++) code, both for spatial and temporal memory flaws. ASan is part of the LLVM clang compiler starting with version 3.1 and a part of GCC starting with version 4.8.
- MemorySanitizer (MSan) MSan adds memory safety checks to C(++) code to detect reading of uninitialised memory. MSan is available as a compile-time option in clang since version 4.0.
- Valgrind Valgrind is the classic instrumentation framework for building tools that do dynamic analysis. It provides several detectors to find different classes of bugs. Valgrind's Memcheck detector for memory errors would be interesting to try for these fuzzing experiments. (Valgrind provides many more detectors and tools, for instance detectors to spot thread errors, tools to profile heap usage, and many more.)

5.2.2.1 Valgrind vs ASan/MSan

Valgrind is much slower than ASan and MSan, and has a much larger memory overhead. ASan does set up an insane amount of virtual memory (20 TB), but does not use all of it. ASan can spot some flaws that valgrind's Memcheck cannot. Conversely, Memcheck can spot some flaws that ASan can't. For instance, ASan does not try to spot access to uninitialised memory, which Memcheck, like MSan, does. So using ASan together with MSan comes closer to what Memcheck does.

Memcheck can detect more problems than MSan. For instance, Memcheck tries to detect memory leaks, which MSan does not. ASan has some support to detect memory leaks, with LeakSanitizer, which is not enabled by default on all systems. For this project detecting memory leaks is probably not interesting: memory leaks typically occur when an application runs for a longer time, not when it is used one-off to process a single input file.

5.2.2.2 afl-cmin and afl-tmin

afl comes with two helper programs that may be interesting to play with: - For some initial set of input files (aka the initial test corpus) that you use to get the fuzzers started, afl-cmin tries to determine the smallest subset that still has the same code coverage as the full set. Code coverage here is measured using the instrumentation that afl adds to a program to observe its control flow and distinguish different execution paths. So, for example, if there are two input files that trigger exactly the same execution path, then afl-cmin will remove one of these files from this test corpus. This will speed up fuzzing, not just for afl but also for other mutation-based fuzzers, as mutating one of these files is likely to generate similar inputs as mutating the other, so having both in the test set is likely to double the amount work for the same outcome. - Given a single input file that triggers a crash, afl-tmin tries to generate a smaller input file that still triggers this crash.

Note that afl-cmin and afl-tmin try very different forms of minimisation: afl-cmin tries to minimise a set of inputs and afl-tmin tries to minimise a single input.

5.2.3 Static analysis tools

- ESBMC
- JBMC
- DSVerifier