



IP PARIS



Hardening System Programming

SSP-RS — Safe System Programming (in Rust)

Samuel Tardieu Stefano Zacchiroli

2024-09-24



How do we find (security) bugs in system programs?



Last week

- What is system programming
- Memory unsafety in system programming
- Real-world security impact of memory unsafety
- (lab) Developer setup and memory unsafety in practice

How do we fix this?

This week:

- Dynamic analysis
 - Testing
 - Code instrumentation
 - Fuzzing
- Static analysis

Coming up next:

- Better tooling (including programming languages)

Testing

Background — Software testing

Definition (Software testing)

*Testing is the process of analyzing a software item to detect the **differences between existing and required conditions** (that is, bugs) and to evaluate the features of the software item.*

— IEEE Guide for Software Verification and Validation Plans (IEEE Std 1059-1993)

- You need a **specification** to determine what is, in fact, a bug
- Specifications consist of multiple **requirements**
 - Requirements can be partitioned in functional and non-functional requirements
 - Functional requirements are about system features and their semantics (the term refers to mathematical functions that relate program inputs to outputs)
- **Security** aspects are usually covered in specifications by **non-functional requirements** (along other aspects such as performances, conformance to standards and regulations, etc.)
- Testing can be performed by **manual** and/or **automated** means

Background — Software testing concepts

System Under Test (SUT)¹ The part of the [software] system that is being tested

Test goal The specific *behavior* of SUT that is being tested

Test input Data provided as input to SUT to trigger the desired behavior

Expected result What SUT *should return* in response to test input

Actual result What SUT *actually returns* in response to test input

Test case A pair `<test input, expected result>`, in the context of a SUT

Test outcome Whether executing SUT with test input returned the expected result or not.
Usually a tristate: OK/KO/failure².

Note that to evaluate test outcomes you need a **test oracle** capable of deciding whether actual result *matches* expected result or not (it can be hard to determine!).

¹also known as Program Under Test (PUT), for pure software systems

²failure = the test case evaluation could not be completed

Background — Testing workflow

- Start from the specification
- For each stated behavior extract a test goal
- For each test goal choose one or more test cases
 - Note that choosing test cases implies **choosing test inputs**
- Encode all test cases into a **test suite**
- Use a **test runner** to run the entire test suite periodically
 - E.g., as a technical requirement for software acceptance,
 - but also at each commit in CI/CD settings

Supporting tooling is available for most programming languages, with good cross-language uniformity (especially if you focus on **unit testing frameworks** adhering to [xUnit](#) conventions).

See, e.g., Wikipedia's [list of unit testing framework](#) for a comprehensive set.

Testing example

Consider the following function to determine whether the length of a string is even or not, together with its specification in the accompanying docstring:

```
/* Takes a string as input. Returns: -1 if the string pointer is NULL; 0 if the string
contains an odd number of characters; a positive integer otherwise. */
int str_is_even(const char *s) {
    int i = 0;
    if (s == NULL) {
        return -1;
    } else {
        for (i = 0; s[i] != '\0'; i++);
        return (i % 2 == 0);
    }
}
```

Q: Which **test cases** would you devise to convince yourself (and others) that the code is correct?


Testing example (cont.)

With, e.g., the [Check](#) unit testing framework we can support xUnit-style testing in C like this:

```
START_TEST ( test_even_strings ) {
    ck_assert_int_eq(str_is_even(""), 1);
    ck_assert_int_eq(str_is_even("qu"), 1);
}
END_TEST
START_TEST ( test_odd_strings ) {
    ck_assert_int_eq(str_is_even("q"), 0);
    ck_assert_int_eq(str_is_even("qux"), 0);
}
END_TEST
START_TEST ( test_null_string ) {
    ck_assert_int_eq(str_is_even(NULL), -1);
}
END_TEST
```

```
$ ./test_is_even # demo-time
Running suite(s): str_is_even
100%: Checks: 3, Failures: 0, Errors: 0
test_is_even.c:7:P:main:test_even_strings:0: Passed
test_is_even.c:13:P:main:test_odd_strings:0: Passed
test_is_even.c:18:P:main:test_null_string:0: Passed
```

How can we choose useful test inputs?

- Information hiding:
 - **Black-box testing:** rely only on the specification, treat the software as something you cannot look *into* to determine test inputs.
 - **White-box testing:** rely (also) on the implementation, e.g., use the source code to choose test inputs that maximize *code coverage*.
- **Edge cases:** given a functionality with a given domain (in the mathematical sense), choose inputs:
 - Within the domain (valid),
 - Outside the domain (invalid),
 - Near either side of the edge of the domain (for both valid and invalid values).
- **Random testing:** choose test inputs randomly at each test case execution.
 - Abundant tooling available to automate this.
 - Problem: non-reproducibility of test execution → inducing **flaky tests**.
- **Mutation testing:** start from a set of predetermined test cases and randomly mutate them to exercise relevant program behavior.
 - Use both heuristics and dynamic fitness analyses to select useful mutants (“killing [useless] mutants”).
 - Active area of [ongoing research](#) .

The problem with testing

Q: Are you convinced our even-string-testing code is correct?

Why or why not?

- Testing is great! Use it always and methodically (e.g., ensure you reach high code coverage of both lines of codes and branching). But.
- Testing is incomplete.
 - **Testing can establish the *presence* of bugs**, but
 - In general **testing cannot establish the *absence* of bugs**.
 - The very best testing could do is showing that the set of *specific* nasty behaviors captured by test cases does not affect your program.

Also, security issues are often correlated with **exceptional circumstances** that you haven't thought of in advance (whereas attackers did!). Hence it is difficult to come up with security-relevant test cases *in advance*. In comparison, identifying relevant test cases for functional requirements is much easier.

(Still, testing is great!)

Dynamic analysis

Dynamic program analysis

- *Software testing* belongs to a larger class of *software verification and validation (V&V)* [↗](#) techniques called **dynamic program analysis**.
- In dynamic program analysis in general we:
 - **Run the program,**
 - **observe what it does...**
 - ...in order to verify that it conforms to its specification
 - (or, more narrowly, that it does *not* exhibit *specific* problematic behavior).

We will now look into common subclasses of dynamic program analysis that can help identifying security vulnerabilities:

- *Binary code instrumentation*
- *Source code instrumentation*
- *Fuzzing*

Binary code instrumentation — Valgrind

Valgrind [↗](#) is a [binary] instrumentation framework for building dynamic analysis tools. There are Valgrind tools that can automatically detect many memory management and threading bugs, and profile your programs in detail. — [Valgrind homepage](#) [↗](#)

- Architecture: Valgrind tool = Valgrind core + tool plugin
- Valgrind core performs **dynamic binary re-compilation**:
 1. Loads your binary executable
 2. Disassembles it (lazily, just-in-time) to an intermediate representation (IR)
 3. Passes IR blocks to the tool plugin for **translation** ← *instrumentation happens here*
 4. Assembles translated IR blocks back into machine code and executes them
- Also Valgrind core:
 - Based on the program execution state S = a finite set of locations that can hold values (= all machine registries + all memory accessible by the program),
 - Maintains a **shadow state** S' of *metadata* about all values in S (e.g., whether the memory where a value resides is initialized or not; the specifics depend on the used tools/plugins).

Memcheck: a memory error detector (based on Valgrind)

Memcheck uses shadow values to track **which bit values are undefined** (i.e., uninitialised, or derived from undefined values) and can thus detect dangerous uses of undefined values. It is used by thousands of C and C++ programmers, and is probably the most widely-used [dynamic binary analysis] tool in existence. — *Nethercote and Seward, PLDI'07*

```
int main() {
    char *buf = (char*)malloc(8);
    buf[16] = 'a';
}
```

(compiler)

```
mov edi, 8
call malloc
mov QWORD PTR [rbp-8], rax

mov rax, QWORD PTR [rbp-8]
add rax, 16
mov BYTE PTR [rax], 97
```

(valgrind)

```
mov edi, 8
call valgrind_malloc
mov QWORD PTR [rbp-8], rax
record memory write ^
```

```
mov rax, QWORD PTR [rbp-8]
record memory read ^
add rax, 16
mov BYTE PTR [rax], 97
record memory write ^
```

Invalid write of size 4
(writing to the heap, but it's not
inside any heap allocation that was
previously made)

Valgrind example

```
1  #include <stdio.h>
2
3  void f(void) {
4      int *x = malloc(10 * sizeof(int));
5      x[10] = 0;
6  }
7
8  int main(void) {
9      f();
10     return 0;
11 }
```

You be Valgrind: what's wrong with this code?

(which builds silently with `--Wall` and runs without segfaulting!)

Valgrind example (cont.)

```
$ gcc -Wall -o valgrind_me valgrind_me.c      # no build-time warnings
$ ./valgrind_me                               # no runtime error
$ valgrind --tool=memcheck --leak-check=yes ./valgrind_me
```

- `--tool=memcheck` chooses Memcheck as Valgrind tool run (which is also the default)
- `--leak-check=yes` asks for details about detected memory leaks, in addition to summary

```
==459318== Memcheck, a memory error detector
==459318== Copyright (C) 2002-2022, and GNU GPL'd, by Julian Seward et al.
==459318== Using Valgrind-3.19.0 and LibVEX; rerun with -h for copyright info
==459318== Command: ./valgrind_me
==459318==
==459318== Invalid write of size 4
==459318==   at 0x109157: f (valgrind_me.c:5)
==459318==   by 0x109168: main (valgrind_me.c:9)
==459318==   Address 0x4a5e068 is 0 bytes after a block of size 40 alloc'd
==459318==   at 0x48407B4: malloc (vg_replace_malloc.c:381)
==459318==   by 0x10914A: f (valgrind_me.c:4)
==459318==   by 0x109168: main (valgrind_me.c:9)
```

[continues...]

Valgrind example (cont.)

```
$ valgrind --tool=memcheck --leak-check=yes ./valgrind_me
[...continues]

==459318== HEAP SUMMARY:
==459318==      in use at exit: 40 bytes in 1 blocks
==459318==    total heap usage: 1 allocs, 0 frees, 40 bytes allocated
==459318==
==459318== 40 bytes in 1 blocks are definitely lost in loss record 1 of 1
==459318==    at 0x48407B4: malloc (vg_replace_malloc.c:381)
==459318==    by 0x10914A: f (valgrind_me.c:4)
==459318==    by 0x109168: main (valgrind_me.c:9)
==459318==
==459318== LEAK SUMMARY:
==459318==    definitely lost: 40 bytes in 1 blocks
==459318==    indirectly lost: 0 bytes in 0 blocks
==459318==    possibly lost: 0 bytes in 0 blocks
==459318==    still reachable: 0 bytes in 0 blocks
==459318==    suppressed: 0 bytes in 0 blocks
==459318==
==459318== For lists of detected and suppressed errors, rerun with: -s
==459318== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)
```

Valgrind — an assessment

(For the specific use case of detecting memory-safety issues.)

Pros:

- Works with **any binary executable** produced by any compiler (you don't even need the source code!). It feels like magic!

Cons:

- Performances: 20-30x **slowdown** w.r.t. original binary (intuition: you are *simulating and verifying* all memory operations)
- Some technical limitations (e.g., kernel code, self-modifying code, etc.)
- Doesn't detect **stack overflows** in the general case: the stack is just a chunk of memory, not handled by a dedicated allocator

(Still, Valgrind is great!)

Valgrind — references

1. Julian Seward, Nicholas Nethercote. *Using Valgrind to Detect Undefined Value Errors with Bit-Precision* [↗](#). USENIX Annual Technical Conference, General Track 2005: 17-30
2. Nicholas Nethercote, Julian Seward. *Valgrind: a framework for heavyweight dynamic binary instrumentation* [↗](#). PLDI 2007: 89-100

First paper is mostly about Memcheck; second one mostly about Valgrind as a generic framework.

Source code instrumentation — LLVM sanitizers

- Same basic idea of Valgrind, but **instrumentation happens at compile time**, when translating from source code to binary code. (The actual verification still happens at runtime though!)
- This way the instrumentation tool can **benefit from additional information** available only at compile-time, e.g., everything related to source code semantics like types, scopes, etc.

Instrumentation tools based on source code instrumentation are often called **sanitizers** (= they check the “health” of your code). [Clang](#)—the C compiler frontend of the [LLVM](#) compiler infrastructure project—comes with a number of built-in sanitizers. E.g.:

[AddressSanitizer](#) memory error detector: out-of-bounds (OOB) accesses (including in the stack!), use-after-free, double free

[LeakSanitizer](#) memory leak detector (often combined with AddressSanitizer)

[MemorySanitizer](#) uninitialized read detector

[ThreadSanitizer](#) data race detector

[UndefinedBehaviorSanitizer](#) detector of undefined behavior situations (as per language specification)

LLVM sanitizers — example

```
1 #include <stdio.h>
2
3 int main(int argc, char** argv) {
4     int *a = new int[10];
5     a[5] = 0;
6     if (a[argc])
7         printf("xx\n");
8     return 0;
9 } // source: https://clang.llvm.org/docs/MemorySanitizer.html
```

```
$ clang++ -fsanitize=memory -g -o sanitize_me sanitize_me.cc
```

```
$ ./sanitize_me
```

```
==468705==WARNING: MemorySanitizer: use-of-uninitialized-value
```

```
#0 0x560150345550 in main [..]/sanitize_me.cc:6:6
```

```
#1 0x7f317af67189 in __libc_start_call_main csu/./sysdeps/nptl/libc_start_call_main.h:58:16
```

```
#2 0x7f317af67244 in __libc_start_main csu/./csu/libc-start.c:381:3
```

```
#3 0x5601502bd2a0 in _start ([..]/sanitize_me+0x222a0) (BuildId: 35a9ca407189643cde6ff5a9[..])
```

```
SUMMARY: MemorySanitizer: use-of-uninitialized-value [..]/sanitize_me.cc:6:6 in main
```

```
Exiting
```

LLVM sanitizers — an assessment

Comparative pros/cons w.r.t. our previous assessment of Valgrind (everything else still applies):

Pros:

- Slightly faster, thanks to compile-time optimizations
- Can detect more issues (e.g., stack overflows) thanks to additional code knowledge

```
int f() {  
    char buf[8];    // Record stack buffer "buf" with size 8  
    buf[16] = 'a'; // Record write to "buf" with offset 16  
}  
  
// ~- this wasn't detectable with Valgrind, as the knowledge that buf is a buffer  
//    allocated on the stack is lost (= hard to determine automatically) in the binary
```

Cons:

- Need source code & recompilation

(Still, sanitizers are great!)

The problem with dynamic analysis

Dynamic analysis can only detect **bad behavior that actually happens** during testing.

- Testing helps, but you will still only find problems if your program exhibits them *on test inputs*.
- (And given in general *program inputs are infinite* you cannot brute force your way out of this.)

How can we find (automatically) **weird edge cases** that trigger bad behavior with high probability?

Fuzzing

Definition (Fuzzing)

*In programming and software development, fuzzing (or fuzz testing) is an automated software testing technique that involves **providing invalid, unexpected, or random data as inputs** to a computer program. The **program is then monitored** for exceptions such as **crashes**, failing built-in code **assertions**, or potential **memory leaks**.* — [Wikipedia](#)

- Semi-automated in most cases, due to the need of providing *bad but well-formed* inputs
- Very-effective, especially for security vulnerabilities: both *memory safety* and *non-sanitized inputs*
 - E.g., what does your program do if you pass 1 MiB or random data as input instead of an integer?
- Can operate on various input sources of a software component, including:
 - *Black box*: CLI arguments, standard I/O streams, filesystem
 - *White box*: function arguments
- Seminal work: class assignment from 1988, showing that 24% of standard UNIX utilities at the time could be crashed by feeding them nasty inputs; see:
Miller et al. [An Empirical Study of the Reliability of UNIX Utilities](#). CACM 33(12): 32-44 (1990)

Fuzzing (cont.)



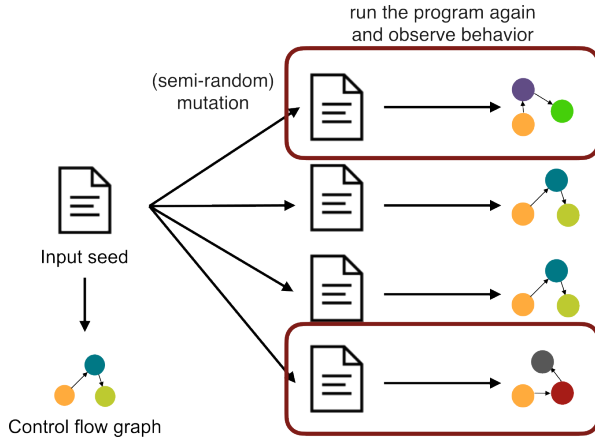
Input seed



Control flow graph

Start from a given (domain-specific and/or random) input, inspect program state at runtime (as with all other dynamic analysis techniques).

Fuzzing (cont.)



Found new behavior, “kill” mutants (= ignore them from now on) that did not lead to new behavior.

Fuzzers

Some open source fuzzers:

[AFL++](#) security-oriented fuzzer, descendant of AFL (American Fuzzy Loop)

[LibFuzzer](#) library-level tester, part of [LLVM](#). Guided by code coverage provided by [LLVM's coverage sanitizer](#).

[zzuf](#) transparent binary fuzzer, intercepting I/O operations (e.g., filesystem, network)

[OSS-Fuzz](#) not a fuzzer *per se*, but a project by Google, Core Infrastructure Initiative, and OpenSSF to continuously fuzz open source software products.

- You can [apply to have your favorite FOSS product fuzzed](#); if selected (Google decides...), the product will be periodically fuzzed and found issues reported back upstream
- Multiple fuzzers used: [LibFuzzer](#), [AFL++](#), [Honggfuzz](#), and [Centipede](#)
- Self-assessment: *“As of June 2021, OSS-Fuzz has found over 30 000 bugs in 500 open source projects.”*

LibFuzzer — example

Let's reconsider our `str_is_even` function:

```
1 int str_is_even(const char *s) {
2     int i = 0;
3     if (s == NULL) {
4         return -1;
5     } else {
6         for (i = 0; s[i] != '\0'; i++);
7         return (i % 2 == 0);
8     }
9 }
```

Q: what's wrong with this code?

Let's find out using LibFuzzer. First, let's add a **fuzz target** function:

```
extern "C" int LLVMFuzzerTestOneInput(const uint8_t *data, size_t size) {
    char buf[size];
    memcpy(buf, data, size);
    str_is_even(buf); // call str_is_even on fuzzed string
    return 0;
}
```


LibFuzzer — example (cont.)

Second: let's fuzz!

```
$ clang++ -g -fsanitize=address,fuzzer -o fuzz_is_even fuzz_is_even.cc
$ ./fuzz_is_even
INFO: Running with entropic power schedule (0xFF, 100).
INFO: Seed: 791004351
[..]
==1016895==ERROR: AddressSanitizer: dynamic-stack-buffer-overflow on address 0x7ffc7fb144a0 at pc 0x55d934426fe3
READ of size 1 at 0x7ffc7fb144a0 thread T0
    #0 0x55d934426fe3 in str_is_even(char const*) fuzz_is_even.cc:10:15
[..]
SUMMARY: AddressSanitizer: dynamic-stack-buffer-overflow fuzz_is_even.cc:10:15 in str_is_even(char const*)
Shadow bytes around the buggy address:
[..]
 0x1000ff5a880: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
=>0x1000ff5a890: ca ca ca ca[cb]cb cb cb 00 00 00 00 00 00 00 00
 0x1000ff5a8a0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
[..]
Shadow byte legend (one shadow byte represents 8 application bytes):
Addressable:          00
Left alloca redzone:  ca
Right alloca redzone: cb
```

LibFuzzer — example (cont.)

Q: what's the right fix?

```
int str_is_even2(const char *s, size_t len) {
    int i = 0;
    if (s == NULL) {
        return -1;
    } else {
        for (i = 0; i < len && s[i] != '\0'; i++);
        return (i % 2 == 0);
    }
}

extern "C" int LLVMFuzzerTestOneInput(const uint8_t *data, size_t size) {
    char buf[size];
    memcpy(buf, data, size);
    str_is_even2(buf, size);
    return 0;
}
```

A: Passing an explicit `len` parameter (which will break API and ABI but cannot be avoided!).

LibFuzzer — example (cont.)

```
$ ./fuzz_is_even2
INFO: Running with entropic power schedule (0xFF, 100).
INFO: Seed: 879222034
INFO: Loaded 1 modules (7 inline 8-bit counters): 7 [0x561e3ff7aea0, 0x561e3ff7aea7),
INFO: Loaded 1 PC tables (7 PCs): 7 [0x561e3ff7aea8,0x561e3ff7af18),
INFO: -max_len is not provided; libFuzzer will not generate inputs larger than 4096 bytes
INFO: A corpus is not provided, starting from an empty corpus
#2 INITED cov: 6 ft: 6 corp: 1/1b exec/s: 0 rss: 32Mb
#3 NEW cov: 6 ft: 8 corp: 2/3b lim: 4 exec/s: 0 rss: 32Mb L: 2/2 MS: 1 InsertByte-
#12 NEW cov: 6 ft: 10 corp: 3/6b lim: 4 exec/s: 0 rss: 32Mb L: 3/3 MS: 4 ShuffleBytes-ChangeBit-Cros
#17 NEW cov: 6 ft: 12 corp: 4/10b lim: 4 exec/s: 0 rss: 32Mb L: 4/4 MS: 5 EraseBytes-InsertByte-Copy
#444 NEW cov: 6 ft: 14 corp: 5/18b lim: 8 exec/s: 0 rss: 32Mb L: 8/8 MS: 2 InsertRepeatedBytes-Insert
#1516 NEW cov: 6 ft: 16 corp: 6/35b lim: 17 exec/s: 0 rss: 32Mb L: 17/17 MS: 2 InsertRepeatedBytes-Insert
#1593 REDUCE cov: 6 ft: 16 corp: 6/34b lim: 17 exec/s: 0 rss: 33Mb L: 16/16 MS: 2 CopyPart-EraseBytes-Insert
#3361 NEW cov: 6 ft: 18 corp: 7/66b lim: 33 exec/s: 0 rss: 33Mb L: 32/32 MS: 3 CopyPart-ChangeBinL
#13170 NEW cov: 6 ft: 20 corp: 8/194b lim: 128 exec/s: 0 rss: 33Mb L: 128/128 MS: 4 ChangeBit-CMP-Copy
#2097152 pulse cov: 6 ft: 20 corp: 8/194b lim: 4096 exec/s: 1048576 rss: 244Mb
#4194304 pulse cov: 6 ft: 20 corp: 8/194b lim: 4096 exec/s: 838860 rss: 456Mb
#8388608 pulse cov: 6 ft: 20 corp: 8/194b lim: 4096 exec/s: 762600 rss: 521Mb
[...]
```

Fuzzing will run for a while this time...

Static analysis

Static program analysis


- With *dynamic* program analysis we execute a program (with instrumentation and/or in a particular simulated environment, etc.) in order to analyze it
- With **static program analysis** the analysis is done *without executing* the program, usually by just looking at the program source code
- Many techniques for doing this exist and decades of CS research have been devoted to them, e.g.:
 - abstract interpretation
 - data-flow analysis
 - formal proofs of program correctness
 - model checking
 - symbolic execution
 - ...
- In this lecture we will just cover the basic idea to understand why it is difficult and how it can help in hardening system programming

Linting

One of the most basic forms of static analysis is called “**linting**” in programming jargon.

Stephen C. Johnson, a computer scientist at Bell Labs, came up with lint in 1978 [...] The term “lint” was derived from the name of the tiny bits of fiber and fluff shed by clothing, as the command should act like a dryer machine lint trap, detecting small errors with big effects. —

[Wikipedia](#) 

- A **linter** verifies the source code of a program against a set of **syntactic rules**
- In its most basic form the linter verifies that the code adheres to a **coding style guide**
 - e.g., from the [Google C++ Style Guide](#) 

All header files should have #define guards to prevent multiple inclusion

```
#ifndef FOO_BAR_BAZ_H_
```

```
#define FOO_BAR_BAZ_H_
```

- Verification is textual (e.g., some form of text search) and **automatic fixing** can be supported as well (e.g., text replacement)

Linting (cont.)

- Linting is most useful for style issues and shines at ensuring that all developers working on the same code base write **uniform code** that does not distract due to visual/style differences
- Linters are generally integrated into tooling (e.g., commit-time hooks) and CI pipelines (e.g., fail the build in case of non-adherence)
- In very limited cases linting can also prevent memory-related bad practices
 - e.g., “do not use `gets()`”, which is trivial to detect textually
 - quite prone to **false positives**, e.g., you can use `strcpy` safely, but many security linters will complain and recommend to use `strncpy` instead

Some linters and linter-related tooling

- [Clang-Tidy](#) for C/C++ (which also does more advanced static analysis, more on this later)
- [flake8](#) and [Black](#) for Python
- [Checkstyle](#) for Java
- ...
- [Gitlint](#) for Git commit messages
- [pre-commit](#): general framework to enforce linting at commit time

Data-flow analysis

- Linting is widely used to enforce coding styles, but not very effective at detecting safety issues.
- Moving toward more powerful techniques for this task, the state-of-the-art of pragmatic static analysis tools is **data-flow analysis**.
 - (Even more powerful techniques exist, such as formal proofs of program correctness, but they are still too expensive and difficult to be widely applicable. They are hence currently used primarily for small and mission critical systems.)

Data-flow analysis

- A static analysis techniques used to **prove facts** about a program or a fragment of it
- Takes into account the **control flow graph** (CFG) of the program (i.e., the order of execution of program instructions)
- Keep track of **all possible values** that variables (or other stateful parts) can take during execution
- Key idea: propagate facts along the edges of the CFG until a fixed point is reached

Data-flow analysis — example 1

```
1 void printToUpper(const char *str) {
2     char *upper = strdup(str);
3     for (int i = 0; str[i] != '\0'; i++) {
4         if(str[i] >= 'a' && str[i] <= 'z') {
5             upper[i] = str[i] - ('a' - 'A');
6         }
7     }
8     printf("%s\n", upper);
9     free(upper);
10 }
11
12 int main(int argc, char *argv[]) {
13     printf("Enter a string to uppercase, or type \"quit\" to quit:\n");
14     char input[512];
15     fgets(input, sizeof(input), stdin); // safely read input string
16     char *toMakeUppercase;
17     if (strcmp(input, "quit") != 0) {
18         toMakeUppercase = input;
19     }
20     printToUpper(toMakeUppercase);
21 }
```

Data-flow analysis — example 1 (cont.)

```
printf("Enter a string to uppercase, or type \"quit\" to quit:\n");
char input[512];
fgets(input, sizeof(input), stdin);
char *toMakeUppercase;
                                // <- toMakeUpperCase = { undef }
if (strcmp(input, "quit") != 0) {
    // <- toMakeUpperCase = { undef }
    toMakeUppercase = input;
                                // <- toMakeUpperCase = { input }
}
                                // <- toMakeUpperCase = { undef, input }
printToUpper(toMakeUppercase);
```

Static analysis result: Error: `printToUpper()` might be called on an uninitialized argument.

Data-flow analysis — example 2

```
1 int main(int argc, char *argv[]) {
2     // Goal: parse the substring between brackets, e.g., "foo [bar]" -> "bar"
3     char *parsed = strdup(argv[1]);
4     char *open_bracket = strchr(parsed, '['); // Find open bracket
5     if (open_bracket == NULL) {
6         printf("Malformed input!\n");
7         return 1;
8     }
9     parsed = open_bracket + 1; // Make result string start after open bracket
10
11     char *close_bracket = strchr(parsed, ']'); // Find the close bracket
12     if (close_bracket == NULL) {
13         printf("Malformed input!\n");
14         return 1;
15     }
16     *close_bracket = '\0'; // Replace close bracket with null terminator
17     printf("Parsed string: %s\n", parsed);
18     free(parsed);
19     return 0;
20 }
```

Data-flow analysis — example 2 (cont.)

```
char *parsed = strdup(argv[1]);
                                // <- parsed = { heap allocation }
char *open_bracket = strchr(parsed, '[');
if (open_bracket == NULL) {
    printf("Malformed input!\n");
    return 1;
    // <- parsed = { heap allocation }, but disappeared local variable!
}
parsed = open_bracket + 1;
char *close_bracket = strchr(parsed, ']');
if (close_bracket == NULL) {
    printf("Malformed input!\n");
    return 1;
}
*close_bracket = '\0';
printf("Parsed string: %s\n", parsed);
free(parsed);
return 0;
```

- *Static* analysis result: Error: possible memory leak.
- Compare with Valgrind: here we know it *might* happen even if we haven't witnessed it; Valgrind would detect that it *does* happen, but only when given the “right” input.

Static analysis with Clang-Tidy

```
void free_me_maybe(void *buf) {
    if (rand() == 1)
        return;
    free(buf);
}
int main() {
    void *buf = malloc(42);
    free_me_maybe(buf);
    return 0;
} // end of random_leak.c
```

```
$ clang-tidy random_leak.c
```

```
random_leak.c:13:2: warning: Potential leak of memory pointed to by 'buf' [clang-analyzer-unix.Malloc]
    return 0;
    ~
random_leak.c:11:14: note: Memory is allocated
    void *buf = malloc(42);
                ~~~~~~
random_leak.c:13:2: note: Potential leak of memory pointed to by 'buf'
    return 0;
    ~
```

Good read on how this gets *much* more complicated in practice: [Data flow analysis: an informal introduction](#) (part of [Clang-Tidy](#) documentation).

Static analysis of an entire build with scan-build

From <https://clang-analyzer.lvm.org/scan-build.html>:

scan-build is a command line utility that enables a user to run the [Clang-Tidy] static analyzer over their codebase as part of performing a regular build (from the command line). scan-build has little or no knowledge about how you build your code. It works by overriding the CC and CXX environment variables to (hopefully) change your build to use a “fake” compiler instead of the one that would normally build your project. This fake compiler executes either clang or gcc (depending on the platform) to compile your code and then executes the static analyzer to analyze your code. This “poor man’s interposition” works amazingly well in many cases and falls down in others.

In practice:

- If you have a properly written standard Makefile that run as `make` to orchestrate a build,
- you can run `scan-build make` instead to have *all* the build source code statically analyzed.
- See for example the Firefox scan-build results maintained by Sylvestre Ledru at <https://sylvestre.ledru.info/reports/fx-scan-build/>.

Static analysis — an assessment

Pros:

- Will detect **issues that do not arise in testing** (manual and/or via automated test suites).
 - This is HUGE! It addresses the main problem of dynamic analysis.

Cons:

- Prone to **false positives**: precisely because it identifies all *potential* issues, static analysis could flag theoretical issues that (for reasons unknown to the static analyzer) cannot occur in practice.
 - Tuning and/or overrides mitigate this problem.
 - The main difficulty for any static analyzer is providing a **good signal/noise ratio** to be worth it.
- Practical limitations: unbound values, loops, programs with many components make the combinatorics of data-flow analysis explode. This results in either very-slow (to impossible) analysis, or cutoffs that reduce the thoroughness of the analysis.
- (Also, recall that in the general case it is impossible to automatically prove that any specific bug is absent from an arbitrary input program.)

Better tooling

Takeaways

- Many techniques and practical tools exist to improve the security and quality of system programs:
 - Unit testing
 - Binary code instrumentation
 - Source code instrumentation
 - Fuzzing
 - Static analysis
- You should **use all of them** systematically, because they make your code more secure.
- There is no silver bullet though, *they are not enough* to secure your system programs.

What's next?

Starting next week we will dive into the Rust memory model, which is capable of *guaranteeing* (unlike all the techniques reviewed today) the absence of *specific classes* of security-related issues (e.g., memory unsafety and data races) at the price of having to program a little bit differently when it comes to memory handling.



Credits

- These slides contain material and ideas reused with permission from lecture 2 of Stanford's course [CS 110L](#) (2022) by Ryan Eberhardt, Armin Namavari, Will Crichton, Julio Ballista, and Thea Rossman.