Software Assurance Tips

A product of the Software Assurance Tips Team[1]

Jon Hood

Monday 27th January, 2025

1 Defensive Development Plans

Updated Friday 24th January, 2025

A Software Development Plan should be well-structured and implement secure principles for the lifecycle of a software product. The plan should be rigid in the areas of security while remaining flexible to changing threats and discoveries. When security needs change, a robust change management board can make sure that the changes do not impact the security of the overall architecture. Suppose that an overzealous developer, eager to eke out the greatest performance of their software, decides to include the compiler flag `-fno-stack-protector` while building the software. This article will walk through the examples of a sound, consistent Software Assurance implementation which protects against such issues using a defense-in-depth perspective.

An Example Issue

Suppose that we are given the example code in Listing 1. This code will be used to show the breadth of areas where the issue can escalate in security impact and examine how each layer of Software Assurance can identify it.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
void vulnF(char *input) {
    char buffer[32];
    strcpy(buffer, input);
    printf("Input copied: %s\n", buffer);
}
int main() {
    char toCopy[100];
    printf("Enter input: ");
    fgets(toCopy, sizeof(toCopy), stdin);
    vulnF(toCopy);
    return 0;
}
```

Listing 1: User Input Buffer Overflow

This example is compiled with the `-fno-stack-protector` flag. It represents a basic string copy overflow.

SA-11(1): Static Code Analysis

Most static analysis tools would be able to identify the issue in the code. Making sure that Static Application Security Testing (SAST) is enabled and enforcing is a good, easy first-line of defense. Using the static code analysis tool `infer`, the issue is detected in Listing 2.

```
$ infer -P --bufferoverrun analyze -- gcc -c test.c
test.c:13: error: Buffer Overrun L2
  Offset: [0, 99] Size: 32 by call to `vulnF`.
```

Listing 2: SAST Detection of Buffer Overrun

SA-11(4): Manual Code Reviews

Manual code reviews, peer reviews, peer programming, and acceptance reviews of code changes create a sense of accountability where developers learn from each other.

SA-11(5): Penetration Testing

Pentesting events should include the software below the system level. A robust pentest event will make sure that the binaries installed on the system under test have proper protections enabled. In the case of this binary, the stack smashing canaries (which get overridden for detecting an overflow) are missing, letting the pentest team know that defense-in-depth protection is missing from it. Listing 3 shows a snippet of one pentesting tool which identifies the missing stack protections.

```
$ checksec --file=test
STACK CANARY      NX            PIE
No canary found   NX enabled    PIE enabled
```

Listing 3: Pentest Detection of Stack Protection

SA-11(8): Dynamic Code Analysis

Several dynamic analysis techniques are able to identify the issue. Debugging and dynamic execution with sufficiently large input results in detection of the issue as shown in Listing 4 using Memcheck with Valgrind. Instrumenting the binary and fuzzing the application is also a form of dynamic analysis, where a good and a bad seed input are run quickly by AFL in Listing 5.

```
$ valgrind --leak-check=yes ./test
==43968== Memcheck, a memory error detector
Enter input: aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
Input copied: aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
==43968== Jump to the invalid address stated on the next line
```

Listing 4: Valgrind Detection of Memory Overlap

```
$ AFL_USE_ASAN=1 afl-clang-fast test.c -o test
[+] Instrumented 2 locations with no collisions (non-hardened, ASAN mode) of which are 0 h
$ afl-fuzz -i in -o out -- ./test
cycles done : 2
corpus count : 2
saved crashes : 1
```

Listing 5: Fuzzing for Crashes

SA-15: Development Process, Standards, and Tools

The tools, explicitly including the **tool options and tool configurations**, must be documented. This includes the compiler and pipeline options used to build production-like binaries. The documentation of permitted tool options allows oversight and visibility into how the products coming out of a pipeline are built. Requiring the documentation of which tools and options are used will help prevent deviations from safe and approved options. A good example for the software development plan, configuration option documentation, or supply chain risk management plan would be, "The production software is built with GCC version 14.2 using compiler flags `-Os -march=corei7 -pipe -Wall -Werror` for the x86_64 architecture using Oracle Linux 9."

When someone adds in potentially unsafe flags like `-fno-stack-protector`, `-fpermissive`, `-gnatp`, `-fno-pie`, `-fno-pic`, `-fno-strict-aliasing`, or even compiling the production library with debug symbols (`-g`), this deviates from the whitelist of permitted flags. Even changing memory alignment (`-fpack-struct`) or changing the math calculation precision (`-ffast-math`) can cause serious and detrimental issues to a program.

There are new compiler flags coming out frequently, and sticking to a whitelist of permitted flags, allowing them to change only with proper change management processes, is the best way to prevent potentially dangerous ones from getting in and being extremely difficult to find. This

leads to an important requirement in Software Assurance: having someone who understands security on the change control board. This person should be able to understand architectural issues as changes come in to support more than just the Intel Core i7 processor, as in the aforementioned example. Flags affecting integral endianness (`-mbig-endian`), structure packing (`-fpack-struct`), floating point precision alignment (`-malign-double`), and processor extensions (`-msse`) may not have immediate security concerns until they affect the portability of the code. Having a dedicated security specialist watching for these types of issues during the architecture and design of application changes helps prevent issues from making it into the supply chain.

SA-16: Developer-Provided Training

Having standards and practices is important, but making sure the developers are trained to abide by those standards brings a level of depth and ownership to issues as they are encountered.

SA-17(3) and SA-17(4): Correspondence

The Application Security and Development Security Technical Implementation Guide (STIG) maps the requirement to have a secure coding standard to CCI-3323 which falls under RMF control SA-17(4). Note that the primary concern for STIGs isn't functionality coding standards, such as the JTA-Army, ADA compliance, and Internationalization standards: the cybersecurity coding standards are the secure coding standards referenced here. The program should implement secure coding standards, like the CERT standards which specify many rules that are violated in our example (such as MEM35-C and STR31-C). Establishing a set of coding standards would forbid this type of code from being created, particularly when combined with the developer training programs.

SR-9: Tamper Resistance and Detection

Application whitelisting, binary signing, source fortification (`-D_FORTIFY_SOURCE`), control flow fortification (`-fcf-protection`), position-independent executables and libraries (`-fPIE` and `-fPIC`), non-executable stacks (`-z,noexecstack`), stronger stack protection canaries (`-fstack-protector-strong` and `-fstack-protector-all`), and many other techniques for preventing a secured, approved binary from being tampered with can be enacted. When a binary deviates from what is expected, it should be detected.

Though easy to tamper with, doing extra, unexpected, additional verification can be useful. Compiling a binary with `-frecord-gcc-switches` will store the compiler flags used by GCC in a comment inside the binary; including a manual check to make sure that this comment is consistent with the development plan's permitted flags provides an extra level of defense-in-depth verification at the cost of exposing how the binary is built to potential attackers if it is not stripped out prior to delivery.

Conclusion

There are many areas where issues caused by an errant compiler flag, and even the errant flags themselves, can be detected. No single point of failure is to blame when supply chain compromises sneak their way into production. Testing the detection mechanisms that are in place with blue team events and cooperative vulnerability assessments can build confidence that more pernicious errors don't manifest in the final product. Bringing the compiler versions, flags, configurations, and options under control of the change management board in RMF control SA-15 can be accomplished in many different ways, and a complete software development plan will document policies for its enforcement.

References

[1] Jon Hood, ed. SwATips. https://www.SwATips.com/.