Uncovering Undefined Behavior

Eric Lubin
eblubin@mit.edu

Jared Wong
jaredw@mit.edu

December 13, 2013

1 Overview

In this project we study the effects of undefined behavior in open source software and its ability to cause unstable code to be optimized out by modern compilers.

Given the sheer number of software systems that contain such errors, with about 50% of Debian packages reporting over 80,000 different warnings, we attempted to begin the process of filtering through these results.

With the help of Stack[3], we analyzed 85 packages, from Debian and other locations, and submitted a total of 18 patches, of which 8 have been accepted thus far, with the rest pending approval. This paper presents a case study into the common mis-practices of many developers and has helped us develop a rule-of-thumb heuristic as to which types of bugs are more or less likely to be vulnerabilities.

In section 2 we review our working threat model and the security implications of undefined behavior, in section 3 we break down the observed bugs into a number of different categories that loosely correspond to the types of bugs Stack can detect, and in section 4 we wrap up and discuss future areas of research.

2 Security Overview

2.1 Threat Model

When accessing the security implications of undefined behavior we assume that an attacker has complete knowledge of the source code. Any optimization unsafe bugs that exist are known to the attacker, and are able to be exploited if possible.

2.2 Threats

Undefined behavior can lead to a number of common security vulnerabilities. In our case study of undefined behavior we observe pointer overflows, buffer overflows, integer overflows, uninitialized data, null pointer dereferences, and infinite loops. In a recent case study from 2010 to mid 2011 [2] of the Linux kernel, these types of errors have been shown to account for up to 70% of the CVE reports.

As compilers increasingly take advantage of undefined behavior [3] and attackers continue to exploit behavior caused by undefined behavior it is important for programmers to not use, intentionally or unintentionally, undefined behavior.

3 Classification of Unstable Code

Generally, the bugs we found due to unstable code fell into several main categories. First, null pointer dereferences were the most common type of bug signaled by Stack but did not alter correctness. Mixed in with these null pointer dereferences were actual programmer errors due to accidentally not dereferencing pointers for which they wanted their value. Second, signed integer operation overflow checks were another common source of developer misconception. Thirdly, there were pointer overflow bugs. And finally, there were a class of miscellaneous bugs related to division by zero, buffer overflow by one, and shift left and right overflow.

3.1 Null Pointer Dereference

The most common type of warning given by Stack was that of the null pointer dereference. As the C standard states [1], dereferencing a null pointer is undefined, so modern compilers use this standard to predict all previously dereferenced pointers are already non-null. Often times, programmers write redundant checks to verify something is non-null. For example, a common practice we found was the tendency for developers to check whether a pointer is null before freeing it. Of course, since the pointer had already been dereferenced previously, the check was immediately optimized out. Of course, the free function accepts NULL pointers as well and therefore we see that this check for non-null acts as almost like a defensive programming behavior, so Stack's warnings do not affect correctness or jeopardize security in any way.

An entirely separate class of errors that Stack was able to pick up on by chance were also categorized as null pointer dereferences. Upon further review, these errors can be best categorized as programmer error. In each case, the developer had a pointer which was pointing some value of interest. Due to carelessness, instead of dereferencing the pointer to find its value, the code showed accidentally checked whether the pointer itself was non-zero. Because the pointer had already been dereferenced, Stack threw a

```
char *last_dot = strrchr(path, '.');
/* check if the strings ends in a period */
if (last_dot && (last_dot + 1 != '\0')) {...}
```

Figure 1: The above code snippet gives an example of a programmer error in SVN. Having forgotten to dereference (last_dot + 1), the compiler optimizes out the check against null. STACK warns the user of this programming error.

warning. This fortuitous static checking ability turned out to be quite useful, helping to find two bugs in SVN and one in Audacity. The fixes for all of them were trivial and only needed a single dereference on the pointer of interest, but it is fascinating to see the power of Stack to pick up on other sorts of programmer errors other than those it was intended for. In Figure 1 below, we outline the programmer error that Stack helped fix that went undetected for years.

3.2 Signed Integer Operation Overflow

According to the C standard [1], the overflow of signed integers results in undefined behavior. This standard allows the compiler to make various sorts of optimizations that it wouldn't normally have been able to make because it assumes signed integer arithmetic cannot overflow. We found that many developers incorrectly go out of their way to try to catch these overflows. Instead of using constants like `INT_MAX`, they assume that an overflow will wrap around to its negative value and check to see if the sum is less than one of the operands. Unfortunately, if they were to check this with DEBUG on and no optimizations, it would probably work, leading to a greater misunderstanding among developers about how to correctly check if a signed operation will overflow.

3.2.1 Exploiting libcurl

In the following example, we see an instance of incorrect signed integer overflow and its subsequent security implications for cURL's `curl_parsedate`. In Figure 2 we see a function that is public to the API for computing the time stamp associated with a given string representation of the date. First, the code parses the date string up to everything but the time zone difference and stores the result into a `t` of type `time_t`. Next, it validates that the time zone is one of many available timezones specified in the header file, or if not that it is within 14 hours ahead or behind of GMT. The developer then is aware of the fact that adding the offset to the original time might overflow the time, Unfortunately, seemingly unaware of the distinction between signed-type overflows and how they are not guaranteed to wrap around in the same way as their unsigned integer counterparts, the developer implements these overflow checks incorrectly. The compiler, on sufficiently high optimization levels, takes advantage of the fact that signed types cannot overflow and then assumes that the addition will not

```
time_t t = /* ... */ ;
/* Add the time zone diff between local time
   zone and GMT. */
long delta = (long)(tzoff!=-1?tzoff:0);
if((delta > 0) && (t + delta < t))
    return -1; /* time_t overflow */
t += delta;
```

Figure 2: The above shows an unstable signed integer check for overflow. Interestingly, The developer has gone out of his way to write this check as evidenced by the comment. Unfortunately this check is invalid and simply a common misconception, and the compiler then simplifies $t+$ delta $< t$ to delta < 0 and thus the compound expression evaluates to false. There is a potential overflow in incrementing `t` by `delta`, which is never checked because the compiler has optimized out the unstable code, and this overflow propagates outwards to any library calling this function without any chance of recovery.

```
char *exploit = "19 Jan 2038 03:14:07 -0200";
time_t time = curl_getdate(exploit, NULL);
```

Figure 3: The above shows a working exploit for the `curl_parsedate` bug. Taking advantage of the undefined behavior of signed integer overflow, the overflow goes undetected due to the unstable overflow check.

converge.With this optimization, checks like $y + 250 < y$ get simplified to false, thereby bypassing the extra security checks that the developer put in for the whole sake of limiting bugs.

In response, we wrote a simple exploit that seeks to take advantage of the bug by overflowing `t` due to the time zone difference. We assumed for the sake of simplicity that we were working on a 32 bit machine so that our exploit date string was a small date, but if we were on a 64 bit machine the exploit would work the same just the date would have to about 290 billion years later. Our exploit, shown in Figure 3, passes in to the `parsedate` function a string that represents the maximum representable timestamp with a signed 32 bit integer: 03:14:07 UTC on Tuesday, 19 January 2038. We then append to this a timezone string such as "-0200." When the code path is executed, t is `INT_MAX` and then 20*60*60 is added to it, which overflows. Since the compiler optimized out the overflow check, this undefined value can propagate outwards to any callers of this of this library function and potentially have security implications for them as well.

To patch this bug, we check instead for overflow by checking for `t < INT_MAX - delta`, which fixes the bug and successfully anticipates the overflow before it happens.

```
while ((line[i] != ':') && (line[i] != '\0')
    && (i < line_size)) { /* ... */ }
```

Figure 4: In this example, taken from GnuTLS, the loop is iterating over the indices of the character buffer line. However, this code has a bug because the check to make sure that the index is in bounds comes after access to the index.

```
do {
  pos++;
  size = read(fd, &buf[pos], 1);
  /* ... */
} while ((buf[pos] != '\n') && (size > 0)
    && (pos < 256));
```

Figure 5: In this check, courtesy of Asunder, the index `pos` is incremented inside of a do while loop, and then a check to make sure the index is in bounds (`pos < 256`) is tacked on to the end of a list of checks in the while clause. The length of `buf` is only 256, so not only will there be an out-of-bounds read in the while condition (`buf[pos]`), but there will also be an out-of-bounds write in the call to `read`.

3.3 Buffer Overflow

A common cause for buffer overflows are out of order checks when iterating over a buffer. In Figure 4 and Figure 5 we see examples of this mistake. In each case an out of bounds location was accessed before a check to make sure that the index being accessed was in bounds. The programmers clearly had good intentions, however, they mixed up the order of their checks.

In the Asunder vulnerability it is possible to exploit the code and force it to corrupt it's memory by ensuring that the file descriptor being read from doesn't stall. Here the file descriptor is simply the output of another third-party program.

4 Conclusion

Overall, we have shown the value that Stack brings to the suite of static checkers available to developers to verify the correctness and stability of their code. Stack can even present warnings to the developer that catch simple careless error where certain pointers are not dereferenced, as was the case with Subversion and Audacity.

Nonetheless, we have noticed the difficulty with discovering legitimate exploits based on these undefined behavior bugs. Amidst countless redundant null pointer dereferences, many such bugs are hidden numerous levels deep from the outwards facing components in these packages. As a result, the ability to propagate an invalid input into such a bug and then exploit this bug is highly challenging.

In the future, we hope to take this research further. In particular, we'd like to analyze more large scale packages such as the llvm, julia, and latex packages. Given the time and processor constraints of this project and the slight difficulty we had in compiling llvm in the first place for using with Stack, we would like to be able to devote more time into such an endeavor. Furthermore, while we liked having access to all the Debian packages having already been analyzed by Stack, we spent a lot of our time building other packages and suppressing compiler warnings instead of analyzing optck reports. In the past week, we have discussed writing a harness into `brew install` in order to get Stack results on a much larger variety of packages. Furthermore, we want to focus on existing bugs to attempt to continue on our quest for exploitable, public facing bugs.

References

[1] ISO/IEC 9899:2011, programming languages - c. ISO/IEC, 2011.

[2] Haogang Chen, Yandong Mao, Xi Wang, Dong Zhou, Nickolai Zeldovich, and M. Fraans Kaashoek. Linux kernel vulnerabilities: State-of-the-art defenses and open problems. ACM, 2011.

[3] Xi Wang, Nickolai Zeldovich, M. Fraans Kaashoek, and Armando Solar-Lezama. Towards optimization safe systems: Analyzing the impact of undefined behavior. SOSP, 2013.