INSTITUTE FOR DEFENSE ANALYSES

Initial Analysis of Underhanded Source Code

David A. Wheeler, *Project Leader*

April 2020

IDA Document
D-13166

INSTITUTE FOR DEFENSE ANALYSES
4850 Mark Center Drive
Alexandria, Virginia 22311-1882

The Institute for Defense Analyses is a nonprofit corporation that operates three Federally Funded Research and Development Centers. Its mission is to answer the most challenging U.S. security and science policy questions with objective analysis, leveraging extraordinary scientific, technical, and analytic expertise.

About This Publication

This work was conducted by the IDA Systems and Analyses Center under contract HQ0034-14-D-0001, Project C5206, "Underhanded Code," for IDA. The views, opinions, and findings should not be construed as representing the official position of either the Department of Defense or the sponsoring organization.

Acknowledgements

Reginald N. Meeson, Jr

For More Information

David A. Wheeler, Project Leader
dwheeler@ida.org, 703-845-6662

Margaret E. Myers, Director, Information Technology and Systems Division
mmyers@ida.org, 703-578-2782

Rigorous Analysis | Trusted Expertise | Service to the Nation

INSTITUTE FOR DEFENSE ANALYSES

IDA Document D-13166

Initial Analysis of Underhanded Source Code

David A. Wheeler, *Project Leader*

Executive Summary

It is possible to develop software source code that appears benign to human review but is actually malicious. In various competitions, such as the Obfuscated V Contest and Underhanded C Contest, software developers have demonstrated that it is possible to solve a data processing problem "with covert malicious behavior [in the] source code [that] easily passes visual inspection." This is not merely an academic concern; in 2003, an attacker attempted to subvert the widely used Linux kernel by inserting underhanded software (this attack inserted code that used = instead of ==, an easily missed, one-character difference).

This paper provides a brief initial look at underhanded source code, with the intent to eventually help develop countermeasures against it. The process was as follows:

- Identify, acquire, and summarize existing public examples of underhanded code (aka maliciously misleading code)—in other words, source code that appears benign but does something malicious instead. I found various sources, including the Obfuscated V Contest, Underhanded C Contest, Underhanded Crypto Contest, Underhanded Rust Contest, and the JavaScript Misdirection Contest.
- Briefly summarize literature related to underhanded code beyond contest information.
- Identify promising mechanisms for countering underhanded code.
- Examine one data set (the Obfuscated V Contest) in more detail to find the specific attack methods and whether or not there are countermeasures that would work (adjusting the promising countermeasures as needed). I then identified a small set of countermeasures and measured their effectiveness on this data set.

This initial work suggests that countering underhanded code is not an impossible task; it appears that a relatively small set of simple countermeasures can significantly reduce the risk from underhanded code. I recommend examining more samples, identifying a recommended set of underhanded code countermeasures, and applying countermeasures in situations where countering underhanded code is important and the benefits exceed their costs.

Contents

1. Introduction....1-1
2. Public Samples of Underhanded Code....2-1
3. Literature....3-1
4. Countermeasures....4-1
   A. Discussion....4-1
   B. Potential Countermeasures....4-2
5. Examination of Obfuscated V Contest Entrants....5-1
   A. Obfuscated V Contest Data Set....5-1
   B. Brief Examination of the Obfuscated V Contest Data Set....5-2
6. Conclusions....6-1
Appendix A. Downloading Samples....A-1
Appendix B. Detailed Analysis of Obfuscated V Entries....B-1
References....R-1
Acronyms and Abbreviations....AA-1

1. Introduction

There are various efforts that try to counter subversion of software used by the U.S. military, including work on Supply Chain Risk Management (SCRM), software assurance (SwA), system evaluations of software security, and higher-level Common Criteria evaluations. Countering subversion in critical software of all kinds typically depends in part on human review.

Unfortunately, it is possible to develop software source code that may appear benign to human review but is actually malicious. In competitions, software developers have demonstrated that it is possible to solve a data processing problem "with covert malicious behavior [in the] source code [that] easily passes visual inspection."[1] This issue was discussed as "maliciously misleading code" in *Fully Countering Trusting Trust through Diverse Double-Compiling* [Wheeler 2009]. This is not merely an academic concern; in 2003, an attacker attempted to subvert the widely used Linux kernel by trying to insert underhanded software. This attack used = instead of ==, an easily missed, one-character difference [Corbet 2003] [Felten 2013].

It may be possible to counter such attacks through simple countermeasures. Examples of such countermeasures include using software reformatters, syntax highlighting, and static analysis tools (including the warnings that some compilers can generate). However, I have not found evidence that someone has tried such countermeasures or measured their effectiveness for countering underhanded code.

This paper provides a brief initial look at underhanded code and suggests ways to develop countermeasures against it. The process was as follows:

- Identify, acquire, and summarize existing public examples of underhanded code (aka maliciously misleading code)—in other words, source code that appears benign but does something malicious instead. I found various sources, including the Obfuscated V Contest, Underhanded C Contest, Underhanded Crypto Contest, Underhanded Rust Contest, and the JavaScript Misdirection Contest.
- Briefly summarize literature related to underhanded code (e.g., winner summaries and [Schrittwieser 2013]).

[1] "The Underhanded C Contest: About page" at http://www.underhanded-c.org/_page_id_2.html

- Identify promising countermeasures to counter underhanded code.
- Examine one data set (the Obfuscated V Contest) in more detail to find the specific attack methods and determine whether or not there are effective countermeasures (adjusting the promising countermeasures as needed). I then identified a small set of countermeasures and measured their effectiveness on this data set.

I expressly excluded obfuscated code (i.e., code that is obviously difficult for a human to understand). There are also various algorithms (including compression, minification, and compilation) that take normal source code and generate results that are more difficult to review in comparison with normal source code.[2] In those cases, human reviewers can immediately report that these are hard to understand and review and use that fact (by itself) as a reason for not trusting the software.

[2] For example, the code at https://github.com/aemkei/jsfuck converts arbitrary JavaScript code into sequences of only six punctuation/mathematical marks: []()!+.

2. Public Samples of Underhanded Code

I identified the following samples of underhanded code for possible analysis:

- Obfuscated V Contest (http://graphics.stanford.edu/~danielh/vote/vote.html). This contest was created by Daniel Horn in 2004 and is the earliest "underhanded" programming contest that I found. Note: I (David A. Wheeler) was one of the entrants. However, I submitted multiple times (the rules did not say you could not) and the contest judges discarded all but the first (easiest) of my entries in each of their categories. I no longer have the rest of my entries, so I will only include the two entries of mine that they judged. All judged entrants are published on the website.
- Underhanded C Contest (http://www.underhanded-c.org/). Per its FAQ, "The Underhanded C Contest is an annual contest to write innocent-looking C code implementing malicious behavior." It was first organized by Scott Craver at the State University of New York at Binghamton (aka Binghamton University). As of this time, it has run from 2005 to 2009 and then from 2013 to 2015. The website discusses the winners, runners-up, and a few other contributors, but does not easily provide the complete set of sample underhanded code. I requested the complete set of entrants, but had not received them at the time of this writing. This contest even has a Wikipedia page (https://en.wikipedia.org/wiki/Underhanded_C_Contest). It was inspired by the previous "Obfuscated V Contest" by Daniel Horn. There are various articles discussing the contest or its winners:
    - [Williams 2016] discusses one of the entrants in the 2015 contest.
    - [Prentice 2015] discusses the 2015 winner.
- Underhanded Crypto Contest (https://underhandedcrypto.com/). As of this time, it has run from 2014 to 2018. The contest website does not directly note the 2018 winners; however, the 2018 winners are presented and discussed in a DefCon 26 presentation [Caudill 2018]. The set of all entries is available on GitHub (https://github.com/UnderhandedCrypto/entries).

- Underhanded Rust Contest (https://underhanded.rs/en-US/)[3]. This contest does not seem to have gone anywhere. The contest was announced on December 15, 2016, and its deadline was extended to March 31, 2017, but no winners or samples have been posted since then.
- JavaScript Misdirection Contest (http://misdirect.ion.land/).[4] [Jaric 2015] announced the winner on September 27, 2015. There were 40 entries, and 34 of those entries were valid. The announcement of the winner included a set of jsfiddle.net links to the entrants and observed that:
    - "Many contestants hid the evil code in a Base64-encoded block, often masked as a seed or key."
    - "Using Image.src as a way to send the key was very common…"
    - "Another trick used by more than [one entry] was to include a link to StackOverflow in a comment. I think that was quite clever, because as a code reviewer (and creator) I am used to [finding] these kind of comments that explain unusual code."
    - "Generally I find it easier to skip over code that has a good comment above it, so I think that is a good trick too."
- Underhanded Solidity Coding Contest (USCC) (https://u.solidity.cc/; details are available at its GitHub site https://github.com/Arachnid/uscc). Solidity is a contract-oriented programming language for writing smart contracts that can be implemented on blockchain platforms such as Ethereum. The announcement of the winners of the first (2017) contest is available at [Johnson 2017], and the complete set of 2017 winners is posted on GitHub at https://github.com/Arachnid/uscc/tree/master/submissions-2017/. The developers of Solidity used the contest results to improve their tooling.
- The "Write a program that makes 2+2=5" discussion on StackExchange at https://codegolf.stackexchange.com/questions/28786/write-a-program-that-makes-2-2-5 shows how to do that in a variety of programming languages.
- The "Underhanded code contest: Not-so-quick sort" (https://codegolf.stackexchange.com/questions/19569/underhanded-code-contest-not-so-quick-sort) is a small underhanded code contest. The goal of this contest was to "Write a program, in the language of your choice, that reads

[3] As of March 27, 2017, this site has become unavailable, but it is still available through the Internet Archive at https://web.archive.org/web/2019*/https://underhanded.rs/en-US/

[4] As of March 27, 2019, this site has become unavailable, but it is still available through the Internet Archive at https://web.archive.org

lines of input from standard input until EOF, and then writes them to standard output in ASCIIbetical order, similar to the sort command-line program. … The underhanded part… is to prove that your favored platform is `better,' by having your program deliberately run much more slowly on a competing platform."

- "April Fools Day!" (https://codegolf.stackexchange.com/questions/114891/april-fools-day) is a small underhanded code contest with a few underhanded code samples. The goal is to "write a program or function which appears to print the first ten numbers of any integer sequence (on OEIS, the answerer may choose which sequence), but instead prints the exact text "Happy April Fool's Day!" if and only if it is run on April 1st of any year."
- The "Underhanded Python" posting (https://gist.github.com/L3viathan/e47d359470d5e18a357c67d9e4328c16) is quite clever. It uses the fact that "//" opens a comment in other languages to fool the reader. It is revealed by syntax coloring but even vim syntax coloring was not obvious enough to immediately reveal the attack.
- The 2003 attack on the Linux kernel source code. An attacker attempted to subvert the Linux kernel in 2003 through underhanded code that used = instead of ==. This is discussed in [Corbet 2003] and [Felten 2013].

3. Literature

Many samples of underhanded code are from contests, and those contests often publish information about their submissions (at least for the winners). In this chapter, I present other sources that discuss or initially appeared to discuss underhanded code:

- Elaine Ou in [Ou 2016] commented on the Underhanded C Contest: "Common tactics include triggering an arithmetic overflow, pointer overwrites, and bad hash values. As a result, the code ends up doing the opposite of what a user might expect from a visual inspection. Last year's winning entry put this line in a single header file:

  ```
  typedef double float_t; /* Desired precision for floating-point vectors */
  ```

  By default, float_t is defined as single precision in math.h. The above file overrides the typedef as double precision. By #include-ing this header file in some C files but not others, the programmer passes an array of 8-byte numbers into a function that expects an array of 4-byte numbers. C interprets each 8-byte number as two 4-byte numbers, leading to an array where every other value is 0."
- [Walker 2005] notes that the Underhanded C Contest was "dominated by a small number of tricks: buffer overflow; arrays bounds violation; and [getting] = and == the wrong way around." The article argues that Java counters many of these problems, and then discusses some approaches to writing underhanded code in Java.
- The language and compiler developers of Solidity used the Solidity contest (for underhanded code) to identify shortcomings of the Solidity language. They have since refined the language and its compiler to help counter underhanded code. [Reitwiessner 2017] discusses this:
  - "Many of the submissions exploited the fact that it was possible to shadow built-ins like now or msg. We already added warnings in such situations shortly before the beginning of the contest. The solution is not yet complete… but we are also working on that."
  - "Another very common theme was to use the fact that it is possible to send Ether to a contract without triggering the fallback function, and

thus bypassing any internal accounting that might be done there… This is a quite tricky problem to tackle by means of the language…"

- [Schrittwieser 2013] discusses covert computation, a related technique for hiding functionality in side effects of microprocessors to hide malicious code within harmless-looking executable code. This technique is focused on fooling automated malware detectors that analyze machine code by exploiting differences between the detector's heuristic model of a machine as compared to the actual machine. Being aware of the difference between the heuristic model and the real machine is important in this case. However, [Schrittwieser 2013] focuses on analyzing machine code by automated malware detection systems, whereas this paper focuses on analyzing source code by human reviewers, so this work is out of scope.

4. Countermeasures

As creating underhanded code is a potent means of attack, it is important to identify countermeasures to underhanded code. In this chapter, we discuss the possibility of countermeasures and list some potential countermeasures.

A. Discussion

In the longer term, it would be valuable to categorize all underhanded code samples available based on the exploitations that they used and then use that categorization to develop maximally general countermeasures to prevent recurrences of similar attacks. Although that would not necessarily prevent all future attacks, it would at least counter known categories of attacks, and many other kinds of attacks are more likely to be detected by human reviewers (once the "easy ways to fool reviewers" are prevented). I did not fully categorize the underhanded code samples; indeed, it was challenging to collect this many of them.

Nevertheless, a brief review of the information available and the collected samples made it clear that many underhanded code samples exploited a relatively small number of issues. In many cases, for example, they exploit known common mistakes that developers already make in the relevant programming language:

- Joe Walker [Walker 2005] notes that many of the underhanded C contest entries exploited buffer overflows, array bounds violations, and misuse of = vs. ==.
- Many samples worked by confusing humans about comments (e.g., misleading humans about where the comments started or having active code embedded in a comment).

  As noted earlier, one underhanded Python example involving comments is especially intriguing. This example contained a misleading comment that would only be misleading to a programmer who knew a language other than Python (Python uses "#" to begin a comment, but many other programming languages use "//"). As most professional programmers know multiple languages, a professional programmer is very likely to be misled by this construct. This suggests a need to consider subtle errors based on constructs in other programming languages.

- Many attacks on the Solidity language involved shadowing built-ins. Many programming languages support various kinds of shadowing, but shadowing can confuse human reviewers. The analysis of Solidity also determined that the Solidity tools could use a Satisfiability Modulo Theories (SMT) solver to detect some attacks so that those attacks would be automatically detected [Reitwiessner 2017].

The Underhanded Crypto Contest had very different kinds of winners where this was not the case. Many of the winners of the Underhanded Crypto Contest exploited highly technical weaknesses in cryptography technology. This suggests that in highly technical and specialized fields, such as cryptography, software reviews must be carried out in depth by specialists in that field. This should not be terribly surprising, and it is a reasonable requirement for important and specialized areas like cryptography.

The Underhanded Rust Contest posted an article arguing for the use of fuzz testing, specifically American Fuzzy Lop (AFL) combined with mechanisms that enable per-function fuzzing and assertions that check if the assertions were met (https://underhanded.rs/blog/2017/03/07/mitigating-underhandedness-fuzzing-your-code.html). However, this advice comes from the only contest without any entrants.

B. Potential Countermeasures

Here are some potential countermeasures:

1. *Use syntax highlighting and/or programming fonts that clearly distinguish characters.*

   Most text editors and integrated development environments (IDEs) used by today's software developers provide syntax highlighting that helps developers identify (through color and font changes) different kinds of text (e.g., which text are comments, which are tokens, which are numbers, and so on). This information can also provide hints to reviewers that something is amiss (e.g., that active code is hiding within a comment).

   Unfortunately, current syntax highlighting methods may be too subtle for many software developers to detect some underhanded code, and some developers are color-blind (making highlighting less likely to be noticed). For example, I loaded the underhanded Python by L3viathan, which depended on misleading comments and did not notice the subtle differences in color that were occurring in the editor. In addition, editor syntax highlighting must be quick and is usually not written under the assumption that the highlighted code is malicious; as a result, underhanded code might be designed to cause the highlighting to work incorrectly.

Thus, although I recommend syntax highlighting, it is probably not enough; it's simply too easy for this measure to fail. If highlighting is to be used as a security measure, the code that implements highlighting code should be reviewed for security to reduce the risk of its being misled. The highlighting should be configured to ensure that previous lines cannot invalidate the highlighting. For example, some tools only look a fixed number of lines backwards, and underhanded code might exploit this.

In addition, when syntax highlighting is used, developers should consider making some distinctions more obvious if they are important. For example, the text editor vim's configuration could be modified to make numbers more distinct with the command: `hi Number cterm=reverse term=reverse gui=reverse`. This command instructs vim to highlight all Number tokens for that language as "reverse video" in all vim display modes, making any Number token stand out. Such a configuration creates a much greater contrast between lowercase "l" and the digit "1" than the usual vim default (where they typically have different colors but those differences might not be as obvious).

Programming fonts (aka coding fonts) are fonts designed for use during software development. These fonts often strive to clearly distinguish symbols that are more readily confused in other fonts (e.g., uppercase "O" with the digit "0" and lowercase "l" with the digit "1"). The same concerns about syntax highlighting also apply to programming fonts. I recommend using programming fonts when reviewing source code that may include underhanded code. However, programming fonts are probably not enough to detect underhanded code, even in cases where they can reveal some difference; it is simply too easy for this measure to fail as well.

2. *Require all comments to be on separate comment-only lines (via reformatting or a tool that checks the source code).*

   This requirement can reveal non-comments masquerading as comments (e.g., one of the Underhanded Python examples). This counters code embedded in comments without the harsher approach of reformatting source code. Some developers would find this requirement annoying, as it forbids short comments on the same line as the code it comments on.

3. *Reformat source code to a standard format not under the attacker's control.*

   Forcible reformatting can reveal attacks such as non-comments hidden in comments, misleading indentation, and similar problems. More generally, a reasonable common format can ease later review by others, even when underhanded code is not considered a risk. In addition, there is at least one open source software reformatting program available for most widely used

programming languages. For example, the GNU `indent` program can automatically reformat C code.

Such reformatting is common in some software development communities. For example, one survey found that 70% of all code in the Go programming language had the format as generated by the "go fmt" code formatter [Gerrand 2013]. [Guest 2016] recommends that software developers using Go should use the "go fmt" formatter "ideally on save and certainly before submitting for review."

A disadvantage of this approach is that some developers may object to the new format of their code. Part of the problem is that although writing an automated code formatter that does *some* kind of reformatting is typically not difficult to do, writing a *good* automated code reformatter can be quite difficult. This matters because most developers and projects will not use a reformatter on their code unless it is a good one. Bob Nystrom [Nystrom 2015] reports that the "hardest program I've ever written… [was a good] automated code formatter. … Getting [great] quality [sufficient so people will use it] means applying pretty sophisticated formatting rules. That in turn makes performance difficult. I knew balancing quality and speed would be hard, but I didn't realize just how deep the rabbit hole went. … [A good formatter must apply] some fairly sophisticated ranking rules to find the best set of line breaks from an exponential solution space."

If getting projects and developers to accept the use of a full reformatter is a serious problem, an alternative might be a reformatter that only changes the format in specific cases. For example, a reformatter could forcibly reformat code so that a line switched to a comment mode will not switch to a non-comment-mode before the end of that line, but the reformatter would otherwise leave the format alone.

A potential problem with code reformatting tools is that they are typically not developed with the assumption that the code they are reformatting is malicious. As a result, the code reformatters may themselves have bugs, and some might be exploitable. For example, see the BUGS section of GNU indent's manual, which notes some of its known bugs [FSF 2008]. It may be wise to recompile code before and after reformatting to ensure that the reformatting did not change its meaning. Note that this can be done even in language implementations that do not compile to machine code (such as typical uses of Java, Python, and JavaScript). In these cases the compilation could be implemented as a translation to bytecode (for Java or Python) or as minification (for JavaScript).

4. *Use compiler warnings and style checkers that perform static analysis to detect misleading or dangerous constructs.*

   Many underhanded code examples depend on common mistakes. Tools such as compilers (with warnings enabled) and style checkers can warn about many common mistakes and thus should be useful in countering some kinds of underhanded code. In some cases, these tools will allow the potentially dangerous construct without a warning, but only after some marking on the construct is added to the source code. This marking signals to both developers and reviewers where a potentially dangerous construct is being used. Many style guides forbid some error-prone constructs, and their rules forbidding such constructs can be enforced by compiler warnings and style checkers.

   There are many style guides, and widely used programming languages typically have at least one style guide available. One especially well-known guide is the set of software development guidelines for the C programming language by the Motor Industry Software Reliability Association (MISRA). This set is called MISRA C, and its goal is to aid in the "consistent [and] safe use of C in critical systems." [MISRA 2012]. There have been some published objections to the older 2004 version of MISRA C based on quantitative analysis. Les Hatton applied the 2004 MISRA C rules to a set of real code and found that the "real to false positive ratio is not much better in MISRA C 2004 than it was in MISRA C 1998 and it is unacceptably low in both" [Hatton 2005]. [Boogerd 2008] raised similar concerns. A longer discussion of MISRA C, through its 2012 version, is presented in [Bagnar 2018], which argues that MISRA C is not intended for "bug finding" but for "error prevention" in critical applications. This paper states that, "the use of MISRA C in its proper context is part of an error prevention strategy which has little in common with bug finding, i.e., the application of automatic techniques for the detection of instances of some software errors. … The deviation process is an essential part of MISRA C: the point of a guideline is not 'You should not do that' but 'This is dangerous: you may only do that if (1) it is needed, (2) it is safe, and (3) a peer can easily and quickly be convinced of both (1) and (2).' [Consider it an effective way] to rule out most C language traps and pitfalls. The attitude with respect to incompleteness is entirely different between the typical audience of bug finders and the typical audience of MISRA C. Bug finders are usually tolerant about false negatives and intolerant about false positives. ... This is not the right mindset for checking compliance with respect to MISRA C: false positives are a nuisance and should be reduced and/or confined as much as possible, but using algorithms with false negatives implies that those in charge of ensuring compliance will

have to use other methods. So, compliance to MISRA C is not bug finding and, of course, finding some, many or even all causes of run-time errors does not imply compliance to MISRA C."

Here are a few specific examples of such rules that could help find underhanded code (some of which are specific to C or C++):

- Require special syntax for easily confused operators (e.g., = vs. == in C and C++). This can be enforced by the compiler or a style checker. This is less of an issue in Java (where in most cases only one is possible), but in languages such as C and C++, this is a significant issue. One approach, which can be enforced by the compiler, is to require "extra" parentheses that are not required by the language syntax when "=" is used instead of "==" inside a conditional, to signal to the compiler and human reviewers that the "=" is intentional (and that further review is warranted). This approach is implemented by the gcc warning flag -Wparentheses (this warning flag is enabled by gcc warning flag -Wall) and is used by the Linux kernel developers.
- Require enforcement of the Software Engineering Institute (SEI) CERT C Coding Standard [SEI 2018] rule EXP19-C, "Use braces for the body of an if, for, or while statement."[5]
- Detect and prevent misleading indentation. The Apple "goto fail; goto fail;" vulnerability has already shown that misleading indentation can pass review and lead to a dangerous vulnerability [Wheeler 2017].
- Detect "dead" code, as reviewers might not realize that some of the code they are reviewing will not be executed.

5. *Use static source code analysis security analyzers (e.g., Fortify, Coverity, and cppcheck).*

   These tools perform detailed security-related analysis to detect security problems, such as some buffer overflows and array out-of-bounds access in languages where these problems can occur.

6. *Forbid unnecessary use of dangerous constructs (e.g., C's #define), or constrain them to reduce their risks.*

[5] The exact text of rule EXP19-C, and a list of some of the tools that implement it, is at https://wiki.sei.cmu.edu/confluence/display/c/EXP19-C.+Use+braces+for+the+body+of+an+if%2C+for%2C+or+while+statement

Here are some examples:

- In C and C++, forbid use of the dangerous #define macro system if there is some other way to express the construct (e.g., by using constants or normal functions). When #defines are used, require that the name must have only uppercase letters (at least one), underscores, and digits; this is the usual convention (as it makes macros stand out), and it also makes it impossible for macros to override keywords (which are lowercase). In addition, require that macros be fully parenthesized (e.g., every parameter use must be in parentheses to prevent surprise expansion). To reduce effort, an exception could be granted to allow macros of simple constants (e.g., #define NAME 0), as these are harder to exploit in underhanded code. Finally, require that every #define macro be carefully reviewed by multiple people to search for errors specifically caused by macro expansion.
- Forbid using the same name in multiple visible scopes. Many languages allow names in an inner scope to "hide" or "shadow" the same name present in an outer scope, even when both scopes are lexically visible. Although doing this has clear *semantics*, it is easy to confuse developers and reviewers when the same name has multiple meanings.
- Forbid the use of keywords or common library built-ins as names for other constructs (such as variable names), or at least require additional markings for their use.
- Carefully review any compiler options (especially those other than warning flags), as such options can change how the source code is interpreted.

7. *Forbid (or at least strongly warn about) the use of characters that can be confused with other characters.*
   - For example, prevent the use of lowercase "l" and uppercase "O" as individual tokens, as they are easily confused with the digits 1 and 0. It might be useful to warn about tokens where swapping these letters with digits could result in a different token or legal number.
   - This could be generalized further to prevent the use of similar-looking Unicode characters beyond ASCII characters (e.g., by requiring that only ASCII characters be used, or that only a certain set of clearly distinguishable characters are allowed in source code). This would

prevent, for example, swapping the Greek uppercase character alpha ("A") with the Latin uppercase character "A".

8. *Require that only "good" characters be allowed in source files (or at least detect when a non-good character occurs).*

   For example, there could be a rule that requires source code be only UTF-8 (or even only ASCII characters) and that the only control characters allowed are newline, carriage return, and tab. In many projects, the tab character is either forbidden or is only allowed as a sequence of zero or more tabs at the beginning of a line; in such projects, this rule could be enforced by tools.

   Many source files are only supposed to contain ASCII characters; if so, that rule should be enforced. A variant would be to require UTF-8, but only allow ASCII outside of comments. It would be possible to allow arbitrary UTF-8 in constant strings, but it might be more challenging to counter underhanded code if this were allowed. If UTF-8 is allowed outside of comments, it might be useful to limit which character groups are permitted (e.g., to prevent swapping the Greek uppercase character alpha ("A") with the Latin uppercase character "A"). If UTF-8 is allowed, the source files should probably be required to use a single normalization method so that the "same" characters will have exactly the same sequence of bytes.

9. *Use runtime memory corruption detection when handling code from memory-unsafe languages (e.g., by using address sanitizer (ASAN)).*

   C and C++ programs are often vulnerable to various memory corruption problems, such as bounds-checking errors, pointer errors, and double-frees. Techniques such as ASAN can detect many kinds of memory problems during program execution. If combined with a good automated test suite, ASAN can detect memory problems throughout the execution of the test suite. Runtime memory corruption detection mechanisms can also be used during production; using ASAN in production can have a significant performance and memory cost when applied to C/C++ programs (it typically halves execution speed), but these costs may be acceptable in some cases.

10. *Use fuzzing with assertion checking, and also enable ASAN when doing so if the language (as used) is not memory-safe.*

    Fuzzing can detect a variety of defects in software. When the language (as used) is not memory-safe, ASAN can help detect memory safety errors. While no guarantee, these methods can still detect certain kinds of defects that might elude other methods.

11. *Test software with good test coverage (at least good branch and statement coverage).*

    Testing can detect a variety of problems. Test coverage measures can help determine if code is untested or poorly tested. Poorly tested code may have good branch and statement coverage, but code with poor branch and statement coverage is, by definition, poorly tested (as many parts of the code never get tested at all).

12. *Force undefined or poorly defined constructs to have safer semantics, or at least detect such constructs.*

    [Regehr 2010] discusses the often surprising impact of undefined behavior in C and C++; if undefined behavior is in the program, then the compiler is allowed to let it do anything—not just produce an unexpected result. This is not a theoretical problem. [Zdrnja 2009] discusses a vulnerability in the Linux kernel caused by undefined behavior, where in the construct "`struct sock *sk = tun->sk; … if (!tun) return POLLERR;`" the "if …" expression is silently removed by the compiler. This silent removal might be surprising to a reviewer, but it is allowed by the C language specification because the expression "`tun->sk`" causes the whole program to have undefined behavior if "`tun`" is `NULL`. [Wang 2012] discusses the problem with undefined behavior, demonstrates some examples, and provides some recommendations.

    In some cases, the gcc and clang compilers can detect and report constructs at compile time if given the flag `-fcatch-undefined-behavior`. They can also generate run-time checks to detect some undefined behavior using the option `-fsanitize=undefined`, but note that detection is not guaranteed.

    This problem can also sometimes be resolved by using compiler flags that cause normally undefined semantics to have much safer defined semantics. Examples of this include the gcc and/or clang flags `-fwrapv` (wrap signed integer overflow), `-fno-strict-overflow`, `-fno-strict-aliasing`, and `-fno-delete-null-pointer-checks`. The Linux kernel, for example, enables several such flags to reduce risks from undefined behavior. I recommend that programs not be written to depend on such behavior; however, enabling these options may reduce the impact of a programming error. In some sense, this could be considered modifying the language specification through an option to reduce the risks from that language.

13. *Modify the programming language and/or its development environment to counter underhanded code or at least the constructs that some underhanded code exploits.*

This is the approach taken by Solidity, as discussed in [Reitwiessner 2017]. Modifying a programming language's specification so that formerly undefined constructs become officially defined could be considered part of this category.

14. *Use a programming language that has fewer misleading constructs and/or is harder to write underhanded code in.*

    [Walker 2005] emphasizes this point, noting that many of the C and C++ constructs exploited by some underhanded code cannot occur in Java. In particular, switching from a memory-unsafe to a memory-safe language can prevent a large number of vulnerabilities. It is typically costly to translate or rewrite substantial code bases into another programming language, but in some cases it may be worth it.

15. *Learn continuously.*

    There should be continuous efforts to look for new techniques for creating underhanded code. Continued underhanded code contests can help with this. These efforts to search for potential problems must be coupled with efforts to update languages, tools, and configurations to counter underhanded code as new techniques are discovered. Over time, this learning process should make it increasingly difficult for attackers to develop underhanded code that can slip through both human reviewers and other countermeasures.

5. Examination of Obfuscated V Contest Entrants

A reasonable first step for determining if some countermeasures would be effective would be to examine, in more depth, a set of underhanded code. As noted earlier, the first known public set of underhanded code is from the Obfuscated V Contest in 2004. As this was the first data set available, it seemed appropriate to start with it. The goal of the programs from this contest was to miscount votes while appearing to count votes correctly; some programs even managed to miscount votes only on Election Day while appearing to be correct. In this chapter, I briefly discuss and analyze the data set. Details are presented in Appendix B.

A. Obfuscated V Contest Data Set

The Obfuscated V data is a set of underhanded code written in C or C++ (mostly in C). The data set includes commentary on each program explaining why the contest organizers believed the program did not work as expected. I used this summary to identify the specific lines of code that were part of the attack.

Note that I looked at this data set as merely samples of underhanded code. I did not do an analysis of any real voting system's security as a whole. For example, it is widely agreed by experts that mandating the use of voter-verified paper ballots is a necessary minimum step for secure voting [Gambhir 2019]. However, issues such as the need for paper ballots were beyond the scope of this paper.

The efforts to find the specific malicious lines of code revealed an interesting problem: even if a human knows the code is malicious, and has a hint about why it is malicious, the problem can be hard to find. In four entries, I found that the original contest summaries were wrong, misleading, or lacking the attack code (I reported these problems to the contest runner, Daniel Horn):

1. Ryan Cumings: The original summary was "Complex code to hide a simple O vs 0 swap." Although the O vs. 0 is present on line 112, this is not the problem that causes the code to incorrectly count votes. Instead, the primary problem is the misuse of the modulus operator "%" on lines 130, 135, 140, and 144, which causes all votes to only apply to rows 0 or 1 of the table "tbl."
2. Travis Fisher: The original summary was "macro madness... replaces the unsigned char Vote with a crazy expression that does some vote skewing (this

is done on the gcc command line).” However, the compiler expression that caused the attack was not originally documented. After I reported the problem, Daniel Horn added the following text:

> Unfortunately since the exact command line was lost to the sands of time, this one will need to suffice: gcc -Disspace="'K'==" macro_tfisher.c

This compiler option subverts the otherwise correct code by redefining isspace (which normally just returns true if the character given is a space) into a comparison to the letter “K” (that is, it becomes an expression that returns true if the character given is a “K”). While this option does subvert the system, it is extremely obvious. Compilers do support definition of macros, but it is suspicious to set a macro named with lowercase letters (by convention only uppercase letters are used), it is very suspicious to use a compiler to define a non-constant value (any value after an “=” would normally be an integer), and it is extremely suspicious to redefine a well-known standard name like isspace through a compiler option.

3. Geir Thomassen: The original summary was “#define of EOF to changing input to bush using comma operator.” The attack code is missing; there is no #define of EOF and no use of the comma error.
4. Brad Grzesiak: The original summary was “gets instead of equals.” There is a function named `gets` in C, but this sample never calls that function. The word “gets” is sometimes used informally to refer to assignment (“=”), and that is probably what was meant. I believe a clearer description of the problem would be “Use of = instead of the correct == in lines 8-10.”

As the published sample from Geir Thomassen (gthomassen) does not have an attack, it was excluded from the analysis. The rest were used after their information was corrected. This left a total of 40 samples of underhanded code for analysis (once the sample by Geir Thomassen was removed).

B. Brief Examination of the Obfuscated V Contest Data Set

After examining the specific vulnerable lines of code for each of the 40 samples, I looked for some common patterns. Certain kinds of underhanded code were especially common. Out of these 40 samples, 5 involved write or read overflow, 9 involved misuse of the C #define macro mechanism, 7 involved character confusion (confusing lowercase “l” with the digit “1” or confusing uppercase “O” with the digit “0”), 2 involved code hidden within a comment, and 2 involved confusion between = (assignment) and == (equality). This means that 62.5% (25/40) of the sample attacks are covered by these

specific problems. This is already encouraging, as it appears that certain kinds of problems repeat (in some cases, many times).

A related observation is that many of these underhanded code samples exploited common mistakes that occur in normal code. For example, switching between = (assignment) and == (equality) is a common well-known mistake many software developers have made. This means that tools designed to detect common mistakes (such as compiler warning flags) might be helpful for detecting underhanded code.

I then selected five simple countermeasures, in part based on this simple categorization:

1. ASN: Is this attack expected to be detected if run using Address Sanitizer (ASAN)?
2. #d: Is this attack expected to be countered if the use of #define is sharply curtailed? The idea is to require that #define (which is easy to misuse) may only be used if the construct cannot be implemented another way (e.g., via const constants and normal functions), and by imposing limits on #define. This could be enforced during a code review.
3. L1: Forbid the use of lowercase "l" and uppercase "O" in a name as these are too similar to the digits "1" and "0." This includes the use as a variable name, constant name, or enumerated value. More generally, this rule should ensure that names in a scope must be distinguishable from each other and from integer constants if lowercase "l" and the digit "1" and uppercase "O" and the digit "0" are arbitrarily swapped. This could be implemented by a tool; for this paper, I simply determined if a tool (if implemented) could find the underhanded code. This approach could be further generalized to prevent the use of similar-looking Unicode characters that could lead to confusion.
4. Co: Require developers to use syntax-coloring text editors (which reveal code in comments) and forcibly reformat code so that once a line has switched to a comment mode, it may not switch to a non-comment-mode before the end of that line.
5. gcc: Process the code through various gcc warning flags to detect potential problems. In particular, run gcc with the options "-Wall -Wextra -pedantic -Wbad-function-cast -Wstrict-prototypes -Wundef -Wshadow -Wlogical-op –Wconversion." This enables a large set of warnings about various potentially dangerous constructs; in particular, "-Wall" includes many other warnings.

I then examined which samples from the Obfuscated V Contest would be detected by which countermeasure. For example, for the "gcc" countermeasure, I ran gcc (with the warning flags as listed above) on each sample to see if gcc provided a relevant warning. I

found that 75% (30/40) of the samples were detected by at least one of these five simple countermeasures (see Appendix B for details). I believe this is a remarkably high percentage of underhanded code that could be caught by such a small set of simple countermeasures.

In fact, 77.5% (31/40) should have been detected by this set of simple countermeasures. The sample by Kristian Nielsen knielsen (file knielsen.c) should have been detected due to its misleading indentation (in this sample, a "while" loop repeated an "if" statement, but the keyword "if" and "while" had the same indentation). This suggests that there is a bug in the implementation of the misleading indentation detector in the version of gcc used. I reported this defect to its developer. This example suggests a larger lesson: if warning flags are to become an important part of security, it is important to have robust test suites for these tools that are run on every change so that they are unlikely to have serious defects. Otherwise, there is a significant risk that the tools will not find the underhanded code they are supposed to find.

Of the 22.5% (9/40) that were not detected using these countermeasures (if they had all worked correctly), I judge that over half (6) of them would be suspicious to human reviewers (e.g., because they use highly nonstandard constructs, unusual formats, or an unusual amount of repetition). This leaves only 7.5% (3/40) code samples that are neither suspicious nor detectable using the simple set of countermeasures. These samples are:

1. mrussotto: Line 18 looks like it initializes the "Tallymap" array, but it ranges over sizeof(Tally) instead of sizeof(Tallymap). As a result, it only initializes a few values instead of all of the values of Tallymap. The attack depends on an unusual initialization value of Tallymap to avoid warnings about uninitialized values, and that could be considered somewhat suspicious. In short, a reviewer might ask, "Why does this code initialize an array to an unusual initial value and then run a loop to initialize it to the correct value?" The answer, of course, is that changing the code to use a correct initial value from the beginning would make the program work correctly. This underhanded code might be detected by more advanced checks to warn when there is a loop over only part of an array or by requiring additional human review of initialization values other than 0 or 1.

2. rseace: Line 24 performs a bad type conversion. It should be possible to add additional compiler warnings to detect bad type conversions like this and report a warning if there is no explicit cast. Underhanded code like this would be countered by such a warning; once this kind of dangerous cast became explicit, it would be far more obvious to a reviewer.

3. jsohn: Line 27 incorrectly checks twice that a value is "n," instead of both "n" and "N," in the expression ((Vote == 'n') || (Vote == 'n')). This could be

detected by a compiler mechanism (or warning) that notices useless operations such as checking for the same condition twice in an OR expression. This would be a generally useful warning, as it could even detect some potential defects in non-underhanded code. In addition, compiler optimizers already look for common subexpressions, so this check might be relatively easy to add to optimizing compilers.

In short, the few countermeasures were effective at countering many underhanded code samples, and I believe additional countermeasures could be developed to counter the rest of the underhanded code samples that were not already suspicious.

On a more personal note, I found that although it was hard to find some of these attacks at first, it became easier over time. This effect could simply be because the samples were sorted from hardest to easiest to find. However, I believe that there is another factor—once a reviewer reads some underhanded code, it becomes easier to find other underhanded code (especially when it uses a similar kind of attack). This suggests that training could be useful. If this effect is valid, then in cases where underhanded code could cause serious damage, reviewers could be trained to look for underhanded code (using samples), and this training might dramatically increase the probability of detecting such code.

It could be claimed that this approach is unfair, as I knew what to look for before selecting countermeasures. However, the goal is to merely demonstrate that it is often possible to detect underhanded code using a small set of simple general countermeasures. For that goal, this is a reasonable approach, and I believe I have demonstrated that it is possible.

6. Conclusions

Underhanded code is a challenge to counter, but this initial effort to collect samples and examine them suggests that it is possible. Indeed, the Solidity underhanded contest has already led to the developers of Solidity to change their system to counter or warn about dangerous constructs. The Solidity example is instructive; they could not necessarily counter everything, but they could reveal many attacks, make others hard to exploit, and document what reviewers should specifically look for.

I performed a simple analysis of the Obfuscated V sample data set. In this analysis, I found that the five selected simple countermeasures were able to detect 75% (30/40) of the sample underhanded code. One was not detected due to an error in one of the tools, and I believe another six would be considered suspicious to reviewers, so only 7.5% (3/40) of the underhanded code would have slipped through these five simple countermeasures and initial review. I have also identified potential countermeasures that I believe would address the remaining underhanded code samples. This suggests that countering underhanded code is by no means an impossible task. Instead, I believe this provides evidence that a relatively small set of simple countermeasures can significantly reduce the risk from underhanded code.

Attackers have an important pair of constraints when creating underhanded code: they must write code that does a specific wrong thing and the code must look like it is doing the right thing. That is a challenging pair of constraints. Today, an attacker's job is much easier because countermeasures are rarely applied with the purpose of countering underhanded code. If a set of broadly useful countermeasures is used to counter underhanded code, the attackers' task is likely to become much harder and their risk of exposure would dramatically increase, reducing the risk to end users.

Underhanded code countermeasures need not be perfect. Instead, they simply have to be good enough to dissuade potential attackers from trying them or, if that fails, good enough to provide a decent likelihood of detection and/or reduced impact. It would probably be best if some of the countermeasures were publicly known, whereas the details of other countermeasures would be kept private. Some countermeasures are obvious, so it would be impractical to try to make all countermeasures private. The publicly known countermeasures would also help others be more aware of underhanded code and help detect many kinds of underhanded code in any software system. Keeping the details of few countermeasures private (while letting the world know that added countermeasures are being used) would ensure that adversaries would not know exactly how to work around the

entire set of countermeasures, increasing the risks to adversaries that they will be discovered. Ideally, those added risks would dissuade them from creating underhanded code in the first place.

The countermeasures I examined in this paper focused on countering underhanded code that exploits issues in programming languages and their environment. Some underhanded code, such as many of the winners of the Underhanded Crypto Contest, focused instead on higher-level weaknesses in the algorithms they implemented. I did not try to examine this underhanded code in detail. However, these winners suggest that at least in highly technical and specialized fields, such as cryptography, software reviews must be carried out in depth by specialists in that field. This should not be terribly surprising, and it is a reasonable requirement for important and specialized areas like cryptography. It may be possible to develop additional countermeasures for these kinds of underhanded code. Even if it is not, if other kinds of countermeasures are developed and deployed, they will reduce the likelihood that a human reviewer would be misled by other tricks and could instead spend time on deeply understanding what the software does.

The next step would be to examine more attack samples to categorize attacks in more detail, create a larger list of countermeasures, and create a larger matrix to show which countermeasure would counter which attack. This would then be used to identify a recommended set of underhanded code countermeasures that should be applied in important situations. Where possible, some of these countermeasures should be widely available (e.g., as extensions to existing open source software compilers), so that those countermeasures can be applied. Until that time, the short list of countermeasures I have developed should be considered by reviewers when the risk of underhanded code is heightened. The final goal would be to ensure that attacks using underhanded code on important systems are unlikely to pass undetected through the development process to end users.

Appendix A.
Downloading Samples

I created a “makefile” to download and extract many samples of underhanded code. By creating a makefile, data sets can be re-downloaded if these sources change or if there is some other problem. One could simply put this information into a file named “makefile” and run “make” (once “make” is installed). This makefile presupposes that the subdirectory “raw” exists and that the program wget is installed.

```
WGET = wget -r --level inf --convert-links --random-wait
WGET_ONE = wget

all: download

tarball:
	tar cvzf underhanded-samples.tar.gz raw/ makefile

# The "raw" subdirectory contains the "mostly-raw" original website contents;
# we assume "raw/" already exists. The contents of the "raw/" subdirectory
# are not *exactly* the same as original websites in all cases
# because we use --convert-links to convert the
# hyperlinks into a version that will work correctly in a local copy.

# To force a reload, just delete the ".t" timestamp file corresponding
# to the data source.

# The "underhanded C" website only shows winners, not all entries.
raw/www.underhanded-c.org.t:
	cd raw; $(WGET) http://www.underhanded-c.org/
	touch "$@"

raw/obfuscated_v.t:
	mkdir -p raw/obfuscated_v
	cd raw/obfuscated_v ; $(WGET) -nH --cut-dirs=2 --no-parent \
	 http://graphics.stanford.edu/~danielh/vote/vote.html
	touch "$@"

raw/underhanded_crypto.t:
	cd raw ; \
	 git clone https://github.com/UnderhandedCrypto/entries \
	 underhanded_crypto
	touch "$@"

raw/underhanded.rs.t:
```

```
	cd raw; $(WGET) https://underhanded.rs/en-US/
	touch "$@"

raw/misdirect.ion.land.t:
	cd raw; $(WGET) http://misdirect.ion.land/
	touch "$@"


# TODO: Not getting the jsfiddle.net files from javahacker, so
# don't really have the samples
# raw/javahacker.t:
#	cd raw; $(WGET) --domains=javahacker.com,jsfiddle.net --level 2 \
#	 https://javahacker.com/the-first-javascript-misdirection-contest/
#	touch "$@"

raw/uscc.t:
	cd raw; git clone https://github.com/Arachnid/uscc
	touch "$@"

# This is somewhat odd - maybe we shouldn't include these.
raw/2-2-5.t:
	cd raw; mkdir -p 2-2-5; cd 2-2-5; \
	 $(WGET_ONE) https://codegolf.stackexchange.com/questions/28786/write-a-
program-that-makes-2-2-5 ; \
	 mv write-a-program-that-makes-2-2-5 write-a-program-that-makes-2-2-
5.html
	touch "$@"

raw/sort.t:
	cd raw; mkdir -p sort; cd sort; \
	 $(WGET_ONE)
https://codegolf.stackexchange.com/questions/19569/underhanded-code-contest-
not-so-quick-sort ; \
	 mv underhanded-code-contest-not-so-quick-sort underhanded-code-contest-
not-so-quick-sort.html
	touch "$@"

raw/upython.t:
	cd raw; mkdir -p upython; cd upython; \
	 $(WGET_ONE)
https://gist.github.com/L3viathan/e47d359470d5e18a357c67d9e4328c16 ; \
	 mv e47d359470d5e18a357c67d9e4328c16 upython.html
	echo 'NOTE: Must manually extract'
	touch "$@"

# Some of the code was hand-extracted into april-fools
raw/april-fools.t:
	cd raw; mkdir -p april-fools; cd april-fools; \
	 $(WGET_ONE) https://codegolf.stackexchange.com/questions/114891/april-
fools-day
	touch "$@"

# Linux kernel attack hand-copied from:
# https://lwn.net/Articles/57143/
```

```
download: \
       raw/www.underhanded-c.org.t \
       raw/obfuscated_v.t \
       raw/underhanded_crypto.t \
       raw/underhanded.rs.t \
       raw/misdirect.ion.land.t \
       raw/uscc.t \
       raw/2-2-5.t \
       raw/sort.t \
       raw/upython.t \
       raw/april-fools.t
```

Appendix B.
Detailed Analysis of Obfuscated V Entries

This appendix provides a detailed analysis of each entry to the Obfuscated V Contest, including the specific lines of code that cause the errors and a discussion of how their attacks could be countered.

This 2004 contest produced helpful commentary on each scored submission (scores.html). Not all submissions were scored, but as the site only posted code for scored submissions, that is all I could use. There were two sub-categories, "Date Specific Programs" (shown first) and "Date Agnostic Programs." The latter group begins with the entry for Michal Zalewski (mzalewski) below. All of the "Date Agnostic Programs" could be exposed by good tests (as they always produce wrong answers in some cases), but these entries still provide helpful examples of tricks an attacker might try.

The following table examines each entry (where each row is an entry). The first few columns describe the contest entries as follows:

- Score: Score per the contest organizers (10 is the maximum possible score).
- Author: Author name.
- File: Filename (the extension may be omitted).
- Original Summary: The original summary from the Obfuscated V Contest.
- My Summary: My summary, including identification of the specific line numbers that cause the code to be malicious.
- SUS: "Suspiciousness." The original Obfuscated V scoring system preferred programs "that [look] most correct," but some of these programs used constructs that would be highly suspicious to a non-novice developer (even without hints such as syntax coloring by an editor). If code looks suspicious, it is not good underhanded code, because suspicious code invites further review regardless of whether the reviewer can quickly see the problem. Examples of suspicious code includes code with a large number of nonsense #define constructs, the use of #define with lowercase names (a nonstandard practice), and highly nonstandard compiler options that seem almost certain to cause problems. An "X" means I believe it is suspicious, while a "-" means I believe it is not. This is a judgment call, but I believe it is an important one;

even if the error is not apparent, code that looks suspicious is much more likely to get additional review in cases where its correctness is vital.

The next columns briefly determine if a given entry fits into certain categories:

- Ov?: Does this attack depend on an overwrite out-of-range read/write, such as a classic buffer overflow?
- #d?: Does this attack depend on misuse of the #define macro?
- L1?: Does the attack depend on confusing lowercase "l" with the digit "1" or uppercase "O" with the digit "0"?
- Co?: Does the attack depend on misleading comments (code in comments)?
- ==?: Does the attack depend on confusing "=" (assignment) with "==" (equality)?

The next columns determine if a particular countermeasure will counter that entry:

- ASN: Is this attack expected to be detected if run using ASAN? ASAN works by monitoring every memory area to determine if it is allowed to be currently used (read or written); it approximately halves the speed of execution, but it may be acceptable if the task is important. Note that, due to lack of time, I did not actually run ASAN on every program, but given how ASAN works, I believe the results would be as shown.
- #d: Is this attack expected to be countered if the use of #define is sharply curtailed? The idea is to require that the use of the #define macro (which is easy to misuse) may only be used if the construct cannot be implemented another way (e.g., by using const constants and normal functions). The name of any macro must have only uppercase letters (at least one), underscores, and digits; this makes macros stand out and also makes it impossible for them to override keywords. Macros must be fully parenthesized (every parameter use must be in parentheses to prevent surprise expansion, and non-trivial whole expressions must also be parenthesized). An exception may be granted for simple constants (e.g., #define NAME 0). Finally, every #define macro must be carefully reviewed by multiple people to search for errors specifically caused by macro expansion.
- L1: Forbid the use of lowercase "l" and uppercase "O" in names as these are too similar to the digits "1" and "0." This includes the use as a variable name, constant name, or enumerated value. More generally, this rule should ensure that names in a scope must be distinguishable from each other and from integer constants if lowercase "l" and the digit "1" and uppercase "O" and the digit "0" are arbitrarily swapped.

- Co: Require developers to use syntax-coloring text editors (which reveal code in comments) and forcibly reformat code so that a line switched to a comment mode may not be switched to a non-comment-mode before the end of that line.
- gcc: Process the code through gcc with many warning flags enabled to detect potential problems. For these experiments, I ran gcc with the options "-Wall -Wextra -pedantic -Wbad-function-cast -Wstrict-prototypes -Wundef -Wshadow -Wlogical-op -Wconversion". This enables a large set of warnings about various potentially dangerous constructs; in particular, "-Wall" enables many other warnings.[6]

  This countermeasure will detect simple misuse of = vs. ==, because "-Wall" enables "-Wparentheses," an option that warns "if parentheses are omitted in certain contexts, such as when there is an assignment in a context where a truth value is expected…" Put another way, "-Wparentheses" requires extra parentheses when a = is used where == would normally be used. These extra parentheses indicate to developers and reviewers that they should look carefully at the construct.

  This countermeasure should also detect misleading indentation, because the option "-Wall" also enables the option "-Wmisleading-indentation" to detect misleading indentation. Note that the option "-Wunreachable-code" was not used; this option no longer works in gcc, so using it would be misleading.

  The table results shown below were created by running gcc version 7.5.0 on Ubuntu 18.

  I used gcc because it is a widely used compiler. There are many other compilers with warning flags (such as clang), and in addition, there are many other static analysis tools that can also detect various problems in code. If the task is important, many of those tools might be used together, but due to limited time, I used only one of them.
- ALL: An "X" (true) is shown in this column if at least one countermeasure detected this underhanded code (that is, if any of the following columns has an "X": ASN:, #d, L1, Co, or gcc).

[6] Note that all of the options here have a leading hyphen-minus and embedded hyphen-minus, where hyphen-minus is ASCII decimal 45 aka Unicode U+002D. Throughout the rest of this appendix only the hyphen-minus is used; if anything symbol appears to be a related but different symbol, such as the N dash "–" or M dash "—," it is a mistake and should be read as hyphen-minus. This is important because only the hyphen-minus is accepted by the programs discussed here.

Many of these underhanded programs make an unrelated common mistake: they use a printf format of “%d” but pass in a value with a different type, usually long int or long unsigned int. In many circumstances, it will work anyway, but the correct printf format to use is “%ld” for a long int and “%lu” for a long unsigned int. The gcc warning flag “-Wformat” warns of this error. Another common mistake in these entries is a failure to #include the appropriate header files; in many circumstances, this works anyway, but gcc would often report a number of warnings (depending on the specifics). I did not count these (correct) reports as finding the underhanded code, as, in practice, a developer might just fix those problems and the result would still be underhanded. I only counted gcc as finding the underhanded code if gcc specifically identified a line with a report that directly revealed the underhanded code.

Score	Author	File	Original Summary	My Summary	SUS	Ov?	#d?	L1?	Co?	==?	ASN	#d	L1?	Co	gcc	ALL
7.0	Paul Parkanzky	pparkanzky	Uses the fact that the string "second" has more words than "first" or "third" to cause a buffer overrun and skew results. Just gives them to Bush, not uninitialized memory like Matt Eastman.	Classic buffer overflow (CWE-120). Detectable with buffer overflow detectors (run-time & maybe static), certainly countered with ASAN. The key problem is line 48, which is an sprintf that prints to an array with inadequate space (as allocated in line 33). Complaining about sprintf (e.g., requiring snprintf instead of just using sizeof(LigMesg)) would do enough to detect it. Run-time mechanisms that create large "red zones" between memory allocations inside the stack would not detect it, but they would make this exploit fail. Aside: Function getDay uses malloc() in some cases and a pointer to constant in others, making it impossible to deallocate its results safely (and thus guaranteeing a memory leak). That's ugly, but by itself that's not a vulnerability. This is detected by gcc warning -Wformat-overflow.	-	X	-	-	-	-	X	-	-	-	X	X
7.0	Jasvir Nagra	jnagra	Uses the fact that the string "second" has more words than "first" or "third" and the fact that the times between 10 and 12am have 2 characters to cause a buffer overrun and skew results. Just gives	Classic buffer overflow (CWE-120). This is a derived version of Paul Parkanzky's code, so similar comments about countermeasures apply. This is detectable with buffer overflow detectors (run-time & possibly static ones). This is certainly countered by ASAN. The key problem is line 68, which is an sprintf that prints to a fixed buffer allocated on line 44. This is detected by gcc warning -Wformat-overflow.	-	X	-	-	-	-	X	-	-	-	X	X

Score	Author	File	Original Summary	My Summary	SUS	Ov?	#d?	L1?	Co?	==?	ASN	#d	L1?	Co	gcc	ALL
			them to Bush, not uninitialized memory like Matt Eastman. (Submitted past deadline, no victory)													
6.7	Matthew Eastman	meastman	pointer arithmetic, stack smashing	CWE-787: Out-of-bounds Write (child of CWE-119: Improper Restriction of Operations within the Bounds of a Memory Buffer). The vulnerability is in function isElectionDay(). Line 44 passes "&now" to isElectionDay as the parameter "givenTime", a pointer to time_t. Line 102 executes "givenTime--", which does not produce "one second earlier" as it might appear. This is a pointer decrement, and thus this decrement produces a pointer to the *earlier* data. It calls "struct tm *localtime_r(const time_t *timer, struct tm *buf);" It's passed "givenTime", which is valid for the type (since it's a pointer to a time), but now it points to the voting data instead of a time. Line 108 then overwrites the value. The comments around 102 and 108 do a good job of fooling the human into believing that what's being done is reasonable (there's really no good reason for the write on line 108, but the comment does a decent job of justifying the code). No problems were found by gcc -Wall -Wextra, because the types are valid.	-	X	-	-	-	-	X	-	-	-	-	X

Score	Author	File	Original Summary	My Summary	SUS	Ov?	#d?	L1?	Co?	==?	ASN	#d	L1?	Co	gcc	ALL
6.0	Jamie Strachan	sov.cpp	Uses shell script to compile a file, run it, morphing it into another file to run and skew the votes.	Shell script manipulation. Human review is likely to be sufficient to reveal this underhanded code, or at least raise significant suspicions, because it uses an extremely unusual approach for compilation and the shell script to do so is obviously obfuscated. In short, this is not well-hidden underhanded code. Its text says it's a "script and a source file" but lines 1-8 use a sequence of "set" commands that don't work on typical command interpreters like bash or dash. This code appears to assume that the default script system is csh, which would be extremely unusual. Lines 1-8 (the script sequence) looks very much like obfuscated code, which gives it away all by itself to a human reviewer. In addition, lines 136-137 expressly give away the "Sleight", which would probably be immediately rejected by a human review.	X	-	-	-	-	-	-	-	-	-	-	-
5.6	Chris Ruppert	cruppert	pushes bush to nader after nov2 deadline by #defining a strange macro that inserts itself into a case statement. #defining break, a language keyword, is pretty giveaway that something is up.	Misuse of #define. This requires compilation to redefine language keyword "break". Human review is likely to be sufficient to reveal this underhanded code. Redefinition of a language keyword is highly unusual and suspicious. Line 54 detects the recommended setting of "break" to 1, undefines it in lines 56-57, and then redefines it in line 65+ in a multi-line definition that splits the keyword across multiple lines to try to hide what it's doing (but isn't very good at it). The gcc compiler	X	-	X	-	-	-	-	X	-	-	-	X

Score	Author	File	Original Summary	My Summary	SUS	Ov?	#d?	L1?	Co?	==?	ASN	#d	L1?	Co	gcc	ALL
				warnings might be enough of a hint, but they are vague enough that I do not count them.												
5.3	Michal Forisek	mforisek	5.3 Michal Forisek: mforisek: Clever political messages, uses #define to change "another number" to number instead of another_number ,when verifying that the number of votes don't exceed the number of registered voters. the call to time could have been hidden better.	Misuse of #define. Human review is likely to be sufficient to reveal this underhanded code, because this has a huge sequence of highly suspicious #defines. Usually #define would be sparing and define names in UPPERCASE, but these have many lower-case definitions (which are suspicious all by themselves). Requiring #define to be sparing, and all-uppercase, would eliminate this attack. Line 11 "#define news" means "news" will be interpreted as empty, line 12 "#define another news" defines "another" as "news" (thus "another" is also considered empty). Line 27 defines "arbitrary" as "*" so the line 39 "arbitrary number" is interpreted as "*number" instead of the more expected "arbitrary_number"; Line 44 uses "another number" instead of the "another_number" used elsewhere, which becomes "number", and note that number has been subtly declared as a *pointer* to an int instead of an int (because it's been declared as *number). Line 49 sets "arbitrary number = 0", so this is really *number=0 (an assignment of the value 0 to whatever the pointer named number points to). Line 15 defines "prime" as "time news", so the later call to it on line 60 is really a call to time().The gcc compiler warnings might be enough of a hint, but they are vague enough that I do not count them.	X	-	X	-	-	-	-	X	-	-	-	X

Score	Author	File	Original Summary	My Summary	SUS	Ov?	#d?	L1?	Co?	==?	ASN	#d	L1?	Co	gcc	ALL
5.0	Scott Craver	scraver	Local variable l gets assigned to 104 on the big day. When l is later used in place of 1 to multiply bush's count by 104.	Confusion of letter "l" with number "1". The use of the single lowercase letters "l" and "o" as variable names is suspicious by itself, since they are easily confused with "1" and "0". Line 53 says "total = total+l;" – note that in this line, the variable total is incremented by lowercase L, not by one. That said, the entire sequence of lines 51-54 are suspicious. It's not clear that this would work on typical systems.	X	-	-	X	-	-	-	-	X	-	-	X
4.3	Corey Edwards	cedwards	#define madness	#define misuse. Human review is likely to be sufficient to reveal this underhanded code; it has a huge sequence of bizarre and unnecessary #defines. Forbidding #define misuse would certainly counter this. See lines 4-15.	X	-	X	-	-	-	-	X	-	-	-	X
3.3	David Wheeler	dwheeler5	Uses different code on election day.	Election-day code runs incorrect code. This shouldn't be too hard for humans to find, but it isn't trivial because it uses nothing sophisticated to hide the problem, and thus will be hard to find with static analysis tools. That said, this remarkable amount of duplication – which makes the underhanded code slightly harder to find – would easily be considered suspicious by itself. Note that 100% test coverage would trivially find the problem. Line 50 checks if "is_official", and if so, uses code on lines 59-60 which use the wrong tally entries (e.g., "N" for Nader).	X	-	-	-	-	-	-	-	-	-	-	-
7.3	Michal Zalewski	mzalewski	Gives 1/10 of the votes to bush. Relies on macro pasting to alter the	Misuse of #define that hides another variable scope with same name. This is an especially sneaky attack and the attack is not obvious to human review, so this is an	-	-	X	-	-	-	-	X	-	-	X	X

Score	Author	File	Original Summary	My Summary	SUS	Ov?	#d?	L1?	Co?	==?	ASN	#d	L1?	Co	gcc	ALL
			apparent meaning of code (suddenly locally declared t hides t in the outer scope)	especially good example of underhanded code. That said, it uses a macro in a way that is totally unjustified; by itself that is a little suspicious (and would be countered by discouraging unnecessary macro use), and any analysis that warned about hiding a variable of the same name would reveal it. Lines 55-70 define a lengthy macro VOTE_AND_CHECK; it is then invoked by line 78 (match case) and line 83 (unmatched case). Line 55 and on set up a pointer "t" to walk the "tally" array of "struct candidates". However, the macro on line 56 follows a special branch 1/10 of the time. The macro on line 57 defines its own pointer "t", so when line 67 increments the vote, it increments the first tally entry instead of the one requested. The gcc warning –Wshadow warns about the shadowing that is the essence of this attack.												
6.3	Kenneth Davis	kdavis	printf returns 0 or 1 depending on debug, so if debug is turned on it returns correct values, other moves values there.	Complex nested use of terniary operator “?:”. This is very odd code, and human review is likely to complain about it. Also: this doesn't seem to produce wrong answers regardless of the value of debug, so this doesn't seem to be a *working* attack as posted (at least in this test environment). Line 19 sets variable "debug" to value of DEBUG (default 1). Line 33 implements the attack, which is a huge Tally[...] += 1 statement where "..." includes multiple ?:, calls to printf, and use of debug. printf returns the number of characters transmitted, so when nothing is output it	X	-	-	-	-	-	-	-	-	-	X	X

Score	Author	File	Original Summary	My Summary	SUS	Ov?	#d?	L1?	Co?	==?	ASN	#d	L1?	Co	gcc	ALL
				returns 0, but when it outputs text it will modify results. The attack is revealed by -Wunused-value (warning that the left-hand operand of comma expression has no effect).												
6.0	Michael Moore	mmoore	Comments obfuscate ?: statement that blocks out index.	Misformatted comment (early termination due to an embedded */). This can be countered by reformatting comments. Line 108 begins a comment. Line 134 ends the comment, but it's formatted to obscure this. Lines 136-144 include live code but formatted to try to appear as a comment. Line 146 *appears* to end the comment. Significant hints about the attack occur in syntax coloring (lines 136-144 are immediately revealed to be code not comments, and line 146 shows a warning in vim because it ends a comment that has already closed). The gcc warning -Wcomment detects this.	-	-	-	-	X	-	-	-	-	X	X	X
6.0	Jean-Philippe Martin	jmartin	query-replace 1 l where l==-1, buffer overrun	Classic buffer overflow (CWE-120). The #defines in lines 6-7 are highly suspicious, and trivially countered by requiring #define to use uppercase. This uses lowercase "l" as a token that is easily misunderstood as "1", counterable by preventing using of lowercase "o" and "l". Line 6-7 use #define to define "l" as really "t", and "t" as "-1"; Line 29 allocates space, but instead of allocating enough with plus one, it uses plus lowercase-L which is really minus 1. Line 57 assigns to greeting, which is now too short (it assigns to element 64 and 65, but the	X	X	-	X	-	-	X	-	X	-	-	X

Score	Author	File	Original Summary	My Summary	SUS	Ov?	#d?	L1?	Co?	==?	ASN	#d	L1?	Co	gcc	ALL
				maximum index allowed is actually 63). Line 58 overwrites greeting[LASTINDEX] with 0, which is now beyond the array. This is very dependent on exploiting the overlap between the "party" array defined on line 27 and the "greeting" array defined on line 29; inserting extra unused space here destroys the attack. Note that the definition of "TRUE" as -1 is very nonstandard in C (bool.h defines true as 1) and is a little suspicious. In general, there are a number of unnecessary #defines. Note: "cromulent" means "acceptable or adequate" (!).													
5.7	Kester Maddock	kmaddock	Arrays begin at 0, not 1; hence uninitialized memory for the initial bush tally	CWE-125: Out-of-bounds Read (child of CWE-119: Improper Restriction of Operations within the Bounds of a Memory Buffer) and CWE-787: Out-of-bounds Write (child of CWE-119: Improper Restriction of Operations within the Bounds of a Memory Buffer). Line 11 does a "#define NUM_CANDIDATES 4". Line 48 declares "unsigned long Tally[NUM_CANDIDATES] = {0}" which allocates index values 0..3 (the legal values) but nothing else. Line 16 does a "#define BUSH 4"; this means that line 77 that does "Tally[BUSH]++;" probably increments something, but that value being increment is outside of the allocated array. Line 13 "#define OTHER 1" makes line 16 look less suspicious. Line 27+ have a lot of useless temp* variables, which are somewhat suspicious. This could be called CWE-457: Use of Uninitialized Variable,	-	X	-	-	-	-	X	-	-	-	-	X

Score	Author	File	Original Summary	My Summary	SUS	Ov?	#d?	L1?	Co?	==?	ASN	#d	L1?	Co	gcc	ALL
				though I wouldn't; the *variable* is initialized, just not at this out-of-bounds index value.												
5.7	Fredrick Oehrstroem	foehrstroem	misparenthesized #define statement causes ! to cancel out with !BUSH and makes it only true if !!BUSH==input	Misused #define. Line 27 says "#define OTHER(v) !BUSH(v) && !KERRY(v) && !NADER(v)" - note the failure to surround the replacing #define expression with (...). Line 33 says "if (!OTHER(input))", however, this expands to "!!BUSH(v) && !KERRY(v) && !NADER(v)". This can quickly countered by style rules that require parentheses for macro replacements, and also by forbidding the use of #define when not required.	-	-	X	-	-	-	-	X	-	-	-	X
5.7	Adam Barth	abarth	#define, Order of Operations	Line 37 (etc.) define macros that are not properly surrounded with (), e.g., "#define isKerry(v) v==Kerry". Lines 65, 67, and 69 invoke the macros with "!", E.g., line 65 has "!isKerry(v)", which would expand to "!v==Kerry" (a highly improbable construct). This has a ridiculous number of #define statements, counterable by limiting their use to "only when must occur," and can also be detected by requiring (...) to prevent unexpected expansions. The gcc warnings provide adequate hints about the real problem.	-	-	X	-	-	-	-	X	-	-	X	X
5.7	Travis Fisher	tfisher	macro madness... replaces the unsigned char Vote with a crazy expression that does some vote skewing (this is	Dangerous compiler options. Note: The code itself is fine, and the "crazy expression" was not originally documented. After I reported the problem, Daniel Horn added the following text: Unfortunately since the exact command line was lost to the sands of time, this one will need to suffice: gcc -Disspace="'K'==" macro_tfisher.c	X	-	X	-	-	-	-	X	-	-	-	X

Score	Author	File	Original Summary	My Summary	SUS	Ov?	#d?	L1?	Co?	==?	ASN	#d	L1?	Co	gcc	ALL
			done on the gcc command line)	This compiler option is highly suspicious and would probably be immediately obvious to any human reviewer.												
5.3	John Waymouth	jwaymouth	Exploits the fact that order in which arguments are evaluated is undefined in C++ (postincrement in call to doTally)	Undefined order of evaluation. This occurs on line 33. This violates SEI rule EXP30-C, “Do not depend on the order of evaluation for side effects” per https://wiki.sei.cmu.edu/confluence/display/c/EXP30-C.+Do+not+depend+on+the+order+of+evaluation+for+side+effects Note that gcc -Wsequence-point warns about this, noting that “operation on ‘i’ may be undefined.”	-	-	-	-	-	-	-	-	-	-	X	X
5.3	Matthew T Russotto	mrussotto	Uninitialized other variable could cause the map from char to int to point to zero for the 'Other' category	Failure to process entire range of an array to be written to. Line 18 *looks* like it initializes the "Tallymap" array, but it ranges over sizeof(Tally) instead of sizeof(Tallymap), so it only initializes a few values instead of all of them. Lines 22-24 initializes specific values, but doesn't set "other", which would be okay if line 18 had a correct range but it didn't. It's not *uninitialized* nor is it out-of-range because line 12 does an initialization of the Tallymap array with an unusual value, the unusual initialization value does make it look somewhat suspicious. This is challenging to find. The attack doesn't always work (the attack doesn't work on Cygwin).	-	-	-	-	-	-	-	-	-	-	-	-
5.3	David Mazieres	dmazieres	local macro variable c hides the outer c. Pasting	#define and shadowed variable. Lines 84-91 defines macro tabulate(val), which says in line 86 "ptrdiff_t c = val"; note that this macro is NOT defined in uppercase (which	-	-	X	-	-	-	-	X	-	-	X	X

Score	Author	File	Original Summary	My Summary	SUS	Ov?	#d?	L1?	Co?	==?	ASN	#d	L1?	Co	gcc	ALL
			causes this variable to be shadowed.	would warn a developer). Line 105 says "tabulate (c);", but since C macros are not hygienic, this doesn't work as expected, it just means that it's "initialized" with its unset value. The gcc warning -Wshadow specifically warns about this, noting that “declaration of ‘c’ shadows a previous local” and pointing out both the use and shadowed declaration.													
5.0	Rachel ParkeHouben	rparke	Utilizes the fact that constructed order matters. Produces a warning in gcc	CWE-457: Use of Uninitialized Variable (due to construction order). On lines 71-74 it initializes private values, but the later initializations use the values of m_total, but m_total itself is being initialized in the same construct and there is no guarantee m_total has an assignment when it is loaded, e.g., by "m_kerry(m_total)". The gcc warning flag –Wreorder warns about this.	-	-	-	-	-	-	-	-	-	-	X	X
5.0	Joshua Hudson	jhudson	Passes in the number of bytes read instead of the byte read, while obscuring it with setjmp and other nasties.	Line 16 says"vcc(read(0, results, 1), results);" - the function "read" returns the *number of bytes* read (which will typically be 1 since that was the count provided, or 0 on end of file), not the value read. The fact that “read” returns the number of bytes has been true since the early 1970s, so it is not clear how many human reviewers would miss this. The code has many ugly constructs that make it hard to understand in general, though that partial obfuscation suggests there is a bigger problem and would almost certainly make this code suspicious to a reviewer. The gcc warning -Wconversion warns about this, reporting	X	-	-	-	-	-	-	-	-	-	X	X

Score	Author	File	Original Summary	My Summary	SUS	Ov?	#d?	L1?	Co?	==?	ASN	#d	L1?	Co	gcc	ALL
				“warning: conversion to ‘int’ from ‘ssize_t {aka long int}’ may alter its value”												
5.0	Ryan Cumings	rcumings	Modulus operator '%' forces all votes to go to table rows 0 or 1 in lines 130-144. Unrelated, complex code to hide a simple O vs 0 swap.	Unnecessary modulus operator (%). The problem is in lines 130, 135, 140, and 144, which all use the construct "tallyCan(& tbl[vote % 2], ...)". However, "vote % 2" uses the modulus operator "%"; the result is that all the votes can only apply to the address of table tbl rows 0 or 1. There's a mess of commented-out code that make the code this harder to read, which could signal a problem with the code. The original summary was wrong; it said “Complex code to hide a simple O vs 0 swap.” Line 112 does have "0ther" instead of "Other", but that is not the primary problem.	X	-	-	-	-	-	-	-	-	-	-	-
4.3	Robert A Seace	rseace	Casts from ulong* to uchar* and then accesses it. Also issues a warning in gcc with appropriate options.	Cast to wrong type. Line 24 is the key problem, a bad type conversion. That line says, "uchar vote = arg1, *totals = arg2", and the second part converts arg2 to the wrong type. Line 48 declares "ulong totals[NUM_CANDIDATES]" (an array on unsigned longs), and line 53 passes a pointer to that array to function count_vote(). Line 22 defines the count_vote second parameter as "void*", which accepts anything and quietly discards the original type. Line 24 casts "totals" to a new (wrong) type. While gcc warns about the first part of line 24, as configured it did not report the key problem of line 24.	-	-	-	-	-	-	-	-	-	-	-	-

Score	Author	File	Original Summary	My Summary	SUS	Ov?	#d?	L1?	Co?	==?	ASN	#d	L1?	Co	gcc	ALL
4.0	Henrik Abelsson	habelsson	#defines to gotos! Considered harmful.	Line 11 has a malicious definition with an embedded goto. This nonsense is unlikely to pass serious human review and preventing unnecessary #define use would counter it. The gcc -pedantic warning flag reports this problem.	X	-	X	-	-	-	-	X	-	-	X	X
4.0	Eric Noyau	enoyau	Uses l instead of 1	Use of variable lowercase letter "l" instead of the digit "1". Line 23 adds using variable "l" (lowercase "L") instead of number 1 in "k=l+k". Line 12 defines variable "l" (lowercase "L"). This code has ugly formatting.	-	-	-	X	-	-	-	-	X	-	-	X
4.0	David Wheeler	dwheeler1	no break after Other case.	No expected break (unexpected fall through) within a case statement. Line 53 is in a case statement and increments other, but doesn't have a "break" following, so it falls and "other" also increments a candidate in lines 54-55. Line 65 does not print the calculated "other" but instead recalculates an "other" that hides the attack. The filename is actually dwheeler1.c. This attack is revealed by gcc -Wimplicit-fallthrough (which is enabled by -Wextra).	-	-	-	-	-	-	-	-	-	-	X	X
3.7	Craig A Rich	crich	Hides a Tally[] = 0 in a comment. Only works on notepad users.	Hidden code in a comment. Lines 27 and 33 have embedded code hidden in a comment. Vim's syntax highlighting system highlights part of line 33 in red, hinting at the problem.	-	-	-	-	X	-	-	-	-	X		X
3.7	Thiago Campos	tcampos	uses wrong ascii value in part of program. Teaches the value of constants in code.	Line 15 says "int B = 64;" but B is ASCII 66, so Bush values won't be displayed. This is likely to be suspicious to a human and might not pass human review, because this is a weird way to handle ASCII values. It is	X	-	-	-	-	-	-	-	-	-		-

Score	Author	File	Original Summary	My Summary	SUS	Ov?	#d?	L1?	Co?	==?	ASN	#d	L1?	Co	gcc	ALL
				highly unusual to directly define values in code instead of using the compiler to look them up, when simple constructs are available to do so. Such an unusual approach would encourage human reviewers to check it. The values also aren't set as constants, which isn't part of the attack but might cause extra scrutiny. Later in line 32, the values ("constants") are not used, but integers that don't match are used instead (75, 66, and 77 are used, but those don't match the numbers above, and they also don't match the correct values 75, 66, and 78).													
3.7	Jared Sohn	jsohn	Typo in nader section checking if Vote is 'n' or 'n' instead of 'n' or 'N': votes delivered to Other. Guess its' writein. Creative alternative comments.	Conditional checks for 'n' or 'n' instead of 'n' or 'N' (capital letter). On line 27, which says "((Vote == 'n') || (Vote == 'n'))", one of the 'n' constants here should be 'N'.	-	-	-	-	-	-	-	-	-	-	-	-
3.7	Philip Willoughby	pwilloughby	Replaces 1 with l	Uses variable lowercase "l" instead of constant digit "1". Line 43 uses lowercase "l" instead of constant 1 ("BushVotes += l;"). This is enabled through the definition of a variable named "l" on line 23. Remarkably, gcc -Wsign-conversion detects this situation because the “conversion to ‘long unsigned int’ from ‘int’ may change the sign of the result”	-	-	-	X	-	-	-	-	X	-	X	X

Score	Author	File	Original Summary	My Summary	SUS	Ov?	#d?	L1?	Co?	==?	ASN	#d	L1?	Co	gcc	ALL
3.7	Oleg Kibirev	okibirev	uses l instead of 1 when adding to bush's tally. (this is what the ++ operator is for ;-))	Uses variable lowercase "l" instead of constant "1"; line 16 uses "b+=l;" instead of "b+=l;".	-	-	-	X	-	-	-	-	X	-	-	X
3.7	Hacksprint	hacksprint	Replaces 1 with l	Uses variable lowercase "l" instead of constant "1". Lines 26, 33, 40, 46 use lowercase "l" instead of constant 1. This uses lowercase "l" instead of constant "1" in *many* places, which is different from entries such as Oleg Kibirev's (okibirev) (which uses lowercase "l" in just one place). The results thus depend greatly on the ASCII value of characters. The results are especially different from expected values, so practically any testing at all (even when the exact correct answers were not considered) would reveal the attack.	-	-	-	X	-	-	-	-	X	-	-	X
3.7	Alan Krueger	akrueger	Replaces 1 with l	Uses lowercase 'l' instead of '1'. Line 28 uses "++c[l]" instead of the correct "++c[1]".	-	-	-	X	-	-	-	-	X	-	-	X
3.7	Kristian Nielsen	knielsen.c	Puts the kerry case in a while statement that looks for spaces.	Misleading indentation, while loop without {...} following lines 21-26 are a while loop, but the later lines have the same indentation, misleading the reader into thinking that the while loop in line 21 saying "while(isspace(next = getchar()))" simply skips spaces. Note that this violates the recommendation of SEI EXP19-C, "Use braces for the body of an if, for, or while statement". Note: gcc's -Wmisleading-indentation *should* have caught this problem, but for some reason it did not. This defect in gcc has	-	-	-	-	-	-	-	-	-	-	-	-

Score	Author	File	Original Summary	My Summary	SUS	Ov?	#d?	L1?	Co?	==?	ASN	#d	L1?	Co	gcc	ALL
				been reported to its developer. This defect suggests it's important to have test cases to verify warning flag functionality if these warning flags are important to security.												
3.3	Derek Warnick	dwarnick.c	Zero instead of O in the code gives Other votes to Kerry	Malicious switch between zero (0) and isolated letter capital O. Line 27 miscounts the vote, incrementing index 0 (zero) instead of index capital O. It is always possible to use index 0, but this misleading code made sense only because line 11 defines an enumerated value capital "O" as well.	-	-	-	X	-	-	-	-	X	-	-	X
3.3	Matti Niemenmaa	mniemenmaa	= instead of ==	Use of = instead of the correct ==. Line 39 has the error. No matter what the previous value of "Vote" was, it is forced to "K" and then counted as "K". This attack is immediately detected by gcc -Wall, which includes "-Wparentheses", with the warning "suggest parentheses around assignment used as truth value" displayed.	-	-	-	-	-	X	-	-	-	-	X	X
3.3	Drew Vogel	dvogel	bitwise and instead of logical and	Use of "&" instead of the "&&" operator. More specifically line 16 uses "&" instead of the "&&" operator. This construct is suspicious anyway, as it's not in a conditional and there's no obvious reason to use either "and" operator in this situation.	X	-	-	-	-	-	-	-	-	-	-	-
3.3	Brad Grzesiak	bgrzesiak	(OFFICIAL SUMMARY INCORRECT) gets instead of equals	Use of = instead of the correct == in lines 8-10. The claimed problem in the official summary is wrong. Detected by gcc –Wall via -Wparentheses.	-	-	-	-	-	X	-	-	-	-	X	X
3.0	Jonathan Drechsler	jdrechsler	Uses lower case 'k'	Lines 15 and 18 use lowercase 'k' to retrieve values, when it should have been uppercase 'K'. This sample had the lowest score in the	X	-	-	-	-	-	-	-	-	-	-	-

Score	Author	File	Original Summary	My Summary	SUS	Ov?	#d?	L1?	Co?	==?	ASN	#d	L1?	Co	gcc	ALL
				contest; it is a relatively obvious mistake to an experienced developer.												

The sample by Geir Thomassen was excluded, since it does not have an attack. It can be summarized as follows:

4.0	Geir Thomassen	gthomassen	#define of EOF to changing input to bush using comma operator.	This is an error in the data set. There is no #define of EOF and no use of the comma operator. The program as posted appears to work correctly and is not malicious. Note: lines 19-22 use printf %d format, which uses int, but the arguments passed are long unsigned int (not always equivalent); this is detected by gcc -Wall.

References

[Bagnar 2018] Bagnar, Roberto, Abramo Bagnara, and Patricia M. Hill, "The MISRA C Coding Standard and its Role in the Development and Analysis of Safety- and Security-Critical Embedded Software", 2018-09-04, https://arxiv.org/abs/1809.00821

[Boogerd 2008] Boogerd, C.J., and L. Moonen, "Assessing the Value of Coding Standards: An Empirical Study, Delft University of Technology", *ICSM 2008 - IEEE International Conference on Software Maintenance*, 2008, https://repository.tudelft.nl/islandora/object/uuid:646de5ba-eee8-4ec8-8bbc-2c188e1847ea

[Caudill 2018] Caudill, Adam, and Taylor Horby, "The Underhanded Crypto(graphy) Contest", Def Con 26, 2018-11-24, https://www.youtube.com/watch?v=SYUBOLIbxPE

[Corbet 2003] Corbet, Jon, "An attempt to backdoor the kernel", LWN.net, 2003-11-06, https://lwn.net/Articles/57135/

[Felten 2013] Felten, Ed, "The Linux Backdoor Attempt of 2003", 2013-10-09, https://freedom-to-tinker.com/2013/10/09/the-linux-backdoor-attempt-of-2003/

[FSF 2008] Free Software Foundation (FSF), *GNU Indent - beautify C code*, 2008-07-23, https://www.gnu.org/software/indent/manual/

[Gambhir 2019] Gambhir, Raj Karan and Jack Karsten, "Why paper is considered state-of-the-art voting technology", *Cybersecurity and Election Interference* series, Brookings Institution, 2019-08-14, https://www.brookings.edu/blog/techtank/2019/08/14/why-paper-is-considered-state-of-the-art-voting-technology/

[Gerrand 2013] Gerrand, Andrew, "go fmt your code", *The Go Blog*, 2013-01-23, https://blog.golang.org/gofmt

[Guest 2016] Guest, Thomas, "Gofmt knows best", *Word Aligned*, 2016-03-07 http://wordaligned.org/articles/gofmt-knows-best

[Hatton 2005] Hatton, Les, *Language subsetting in an industrial context: a comparison of MISRA C 1998 and MISRA C*, 2005-11-20, https://www.leshatton.org/Documents/MISRA_comp_1105.pdf

[Jaric 2015] Jaric, Peter, 2015-09-27, https://javahacker.com/the-first-javascript-misdirection-contest

[Johnson 2017] Johnson, Nick, Announcing the winners of the first Underhanded Solidity Coding Contest, 2017-09-21, https://medium.com/@weka/announcing-the-winners-of-the-first-underhanded-solidity-coding-contest-282563a87079

[MISRA 2012] Motor Industry Software Reliability Association (MISRA), MISRA C : 2012 (description), 2012, https://www.misra.org.uk/

MISRAHome/MISRAC2012/tabid/196/Default.aspx

[Nystrom 2015] Nystrom, Bob, "The Hardest Program I've Ever Written", *Journal* (blog series), http://journal.stuffwithstuff.com/2015/09/08/the-hardest-program-ive-ever-written/

[Ou 2016] Ou, Elaine, "Obfuscated Obfuscation", (2016-06-07) <https://elaineou.com/2016/06/07/obfuscated-obfuscation/>

[Prentice 2015] Prentice, Lynn (lprent), "Scary programmer", *The Standard*, 2015-06-18, https://thestandard.org.nz/scary-programmer/. Note: the author uses the pseudonym "lprent" in the article; full name shown at https://twitter.com/lprent

[Regehr 2010] Regehr, John, "A Guide to Undefined Behavior in C and C++, Part 1", 2010-07-09, https://blog.regehr.org/archives/213

[Reitwiessner 2017] Reitwiessner, Christian, "Lessons Learnt from the Underhanded Solidity Contest", *Medium*, 2017-09-22, https://medium.com/@chriseth/lessons-learnt-from-the-underhanded-solidity-contest-8388960e09b1

[Schrittwieser 2013] Schrittwieser, Sebastian, Stefan Katzenbeisserz, Peter Kiesebergy, Markus Hubery, Manuel Leithnery, Martin Mulazzaniy, and Edgar Weipply. "Covert Computation — Hiding code in code through compile-time obfuscation." ASIA CCS 2013, Hangzhou, China. https://www.sba-research.org/wp-content/uploads/publications/p529-schrittwieser.pdf

[SEI 2018] Software Engineering Institute (SEI), *SEI CERT C Coding Standard*, 2018, https://wiki.sei.cmu.edu/confluence/display/c/SEI+CERT+C+Coding+Standard

[Walker 2005] Walker, Joe, "Writing malicious code in Java" , 2005-09-28, http://incompleteness.me/blog/2005/09/28/writing-malicious-code-in-java/

[Wang 2012] Wang, Xi, Haogang Chen, Alvin Cheung, Zhihao Jia, Nickolai Zeldovich, and M. Frans Kaashoek, "Undefined Behavior: What Happened to My Code?", *APSys '12*, 2012, Association of Computing Machinery (ACM), https://pdos.csail.mit.edu/papers/ub:apsys12.pdf

[Wheeler 2009] Wheeler, David A, *Fully Countering Trusting Trust through Diverse Double-Compiling*, 2009, PhD Dissertation for George Mason University (GMU), https://dwheeler.com/trusting-trust/

[Wheeler 2017] Wheeler, David A, "The Apple goto fail vulnerability: lessons learned", *Learning from Disaster*, 2017-01-27, https://dwheeler.com/essays/apple-goto-fail.html

[Williams 2016] Williams, Chris, "Winning Underhand C Contest code silently tricks nuke inspectors" https://www.theregister.co.uk/2016/02/04/underhand_c_2015/

[Zdrnja 2009] Zdrnja, Bojan, A new fascinating Linux kernel vulnerability, 2009-07-17, https://isc.sans.edu/diary/A+new+fascinating+Linux+kernel+vulnerability/6820

Acronyms and Abbreviations

ASAN	Address Sanitizer
FAQ	Frequently Asked Questions
IDE	Integrated Development Environment
SCRM	Supply Chain Risk Management
SEI	Software Engineering Institute
SMT	Satisfiability Modulo Theories
SwA	Software Assurance
USCC	Underhanded Solidity Coding Contest

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE (DD-MM-YY)	2. REPORT TYPE	3. DATES COVERED (From – To)
00-04-20	Final	

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
Initial Analysis of Underhanded Source Code	HQ0034-14-D-0001
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBERS

6. AUTHOR(S)	5d. PROJECT NUMBER
David A. Wheeler	C5206
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESSES	8. PERFORMING ORGANIZATION REPORT NUMBER
Institute for Defense Analyses 4850 Mark Center Drive Alexandria, VA 22311-1882	D-13166

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR'S / MONITOR'S ACRONYM
Institute for Defense Analyses 4850 Mark Center Dr., Alexandria, VA 22311	IDA
	11. SPONSOR'S / MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES

Project Leader: David A. Wheeler

14. ABSTRACT

It is possible to develop software source code, called underhanded code, that appears benign to human review but is actually malicious. This is not merely an academic concern; in 2003, an attacker attempted to subvert the widely used Linux kernel by inserting underhanded software. This paper provides a very brief initial look at underhanded source code, with the intent to eventually help develop countermeasures against it. This paper identifies and summarizes public examples of underhanded code, briefly summarizes the literature, and identifies promising countermeasures. It then examines one data set (the Obfuscated V Contest), tries a small set of countermeasures, and measures their effectiveness. This initial work suggests that a small set of countermeasures can significantly reduce the risks from underhanded code. The paper concludes with recommendations on how to expand on this work.

15. SUBJECT TERMS

Underhanded code, underhanded source code, maliciously misleading code, security, software security, computer security, software assurance, malware, subversion, attack, adversary, countermeasures, Obfuscated V, Underhanded C, underhanded, misdirection, software development, syntax highlighting, reformatter, compiler warnings, style checkers, static analysis, static source code analysis, security analyzer, C, address sanitizer, ASAN, software testing, undefined behavior, voting software, elections, software development, DevSecOps, SecDevOps, tools, software tools

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Institute for Defense Analyses
Unclassified	Unclassified	Unclassified	Unlimited	56	19b. TELEPHONE NUMBER (Include Area Code)