Trojan Source: Invisible Vulnerabilities

Nicholas Boucher
University of Cambridge
Cambridge, United Kingdom
nicholas.boucher@cl.cam.ac.uk

Ross Anderson
University of Cambridge
University of Edinburgh
ross.anderson@cl.cam.ac.uk

***Abstract*—We present a new type of attack in which source code is maliciously encoded so that it appears different to a compiler and to the human eye. This attack exploits subtleties in text-encoding standards such as Unicode to produce source code whose tokens are logically encoded in a different order from the one in which they are displayed, leading to vulnerabilities that cannot be perceived directly by human code reviewers. 'Trojan Source' attacks, as we call them, pose an immediate threat both to first-party software and supply-chain compromise across the industry. We present working examples of Trojan-Source attacks in C, C++, C#, JavaScript, Java, Rust, Go, and Python. We propose definitive compiler-level defenses, and describe other mitigating controls that can be deployed in editors, repositories, and build pipelines while compilers are upgraded to block this attack.**

***Index Terms*—security, compilers, encodings**

I. Introduction

What if it were possible to trick compilers into emitting binaries that did not match the logic visible in source code? We demonstrate that this is not only possible for a broad class of modern compilers, but easily exploitable.

We show that subtleties of modern expressive text encodings, such as Unicode, can be used to craft source code that appears visually different to developers and to compilers. The difference can be exploited to invisibly alter the logic in an application and introduce targeted vulnerabilities.

The belief that trustworthy compilers emit binaries that correctly implement the algorithms defined in source code is a foundational assumption of software. It is well-known that malicious compilers can produce binaries containing vulnerabilities [1]; as a result, there has been significant effort devoted to verifying compilers and mitigating their exploitable side-effects. However, to our knowledge, producing vulnerable binaries via unmodified compilers by manipulating the encoding of otherwise non-malicious source code has not so far been explored.

Consider supply-chain attacks which seek to inject vulnerabilities into software upstream of the ultimate targets, such as the recent Solar Winds incident [2]. Two methods an adversary may use to accomplish such a goal are suborning an insider to commit vulnerable code into software systems, or equivalently hacking a computer that they use to do this, and contributing subtle vulnerabilities into open-source projects. In order to prevent or mitigate such attacks, it is essential for developers to perform at least one code or security review of every submitted contribution. However, this critical control may be bypassed if the vulnerabilities do not appear in the source code displayed to the reviewer, but are hidden in the encoding layer underneath.

Such an attack is quite feasible, as we will hereafter demonstrate.

In this paper, we make the following contributions.

- We define a novel class of vulnerabilities, which we call Trojan-Source attacks, which use maliciously encoded, semantically permissible source code modifications to introduce invisible software vulnerabilities.
- We provide working examples of Trojan-Source vulnerabilities in C, C++, C#, JavaScript, Java, Rust, Go, and Python.
- We describe definitive defenses that must be employed by compilers, as well as other defenses that can be used in editors, repositories, and build pipelines.
- We raise a new question about what it means for a compiler to be trustworthy.

II. Background

A. Compiler Security

Compilers translate high-level programming languages into lower-level representations such as architecture-specific machine instructions or portable bytecode. They seek to implement the formal specifications of their input languages, deviations from which are considered bugs.

Since the 1960s [4], researchers have investigated formal methods to mathematically prove that a compiler's output correctly implements the source code supplied to it [5], [6]. Many of the discrepancies between source code logic and compiler output logic stem from compiler optimizations, about which it can be difficult to reason [7]. These optimizations may also cause side-effects that have security consequences [8].

B. Text Encodings

Digital text is stored as an encoded sequence of numerical values, or code points, that correspond with visual glyphs according to the relevant specification. While single-script

TABLE I
UNICODE DIRECTIONALITY FORMATTING CHARACTERS RELEVANT TO REORDERING ATTACKS.
SEE BIDI SPECIFICATION FOR COMPLETE LIST [3].

Abbreviation	Code Point	Name	Description
LRE	`U+202A`	Left-to-Right Embedding	Try treating following text as left-to-right.
RLE	`U+202B`	Right-to-Left Embedding	Try treating following text as right-to-left.
LRO	`U+202D`	Left-to-Right Override	Force treating following text as left-to-right.
RLO	`U+202E`	Right-to-Left Override	Force treating following text as right-to-left.
LRI	`U+2066`	Left-to-Right Isolate	Force treating following text as left-to-right without affecting adjacent text.
RLI	`U+2067`	Right-to-Left Isolate	Force treating following text as right-to-left without affecting adjacent text.
FSI	`U+2068`	First Strong Isolate	Force treating following text in direction indicated by the next character.
PDF	`U+202C`	Pop Directional Formatting	Terminate nearest LRE, RLE, LRO, or RLO.
PDI	`U+2069`	Pop Directional Isolate	Terminate nearest LRI or RLI.

specifications such as ASCII were historically prevalent, modern text encodings have standardized[1] around Unicode [9].

At the time of writing, Unicode defines 143,859 characters across 154 different scripts in addition to various non-script character sets (such as emojis) plus a plethora of control characters. While its specification provides a mapping from numerical code points to characters, the binary representation of those code points is determined by which of various encodings is used, with one of the most common being UTF-8.

Text rendering is performed by interpreting encoded bytes as numerical code points according to the chosen encoding, then looking up the characters in the relevant specification, then resolving all control characters, and finally displaying the glyphs provided for each character in the chosen font.

C. *Supply-Chain Attacks*

Supply-chain attacks are those in which an adversary strives to introduce targeted vulnerabilities into deployed applications, operating systems, and software components [10]. Once published, such vulnerabilities are likely to persist within the affected ecosystem even if patches are later released [11]. Following a number of attacks that compromised multiple firms and government departments, supply-chain attacks have gained urgent attention from the US White House [12].

Adversaries may introduce vulnerabilities in supply-chain attacks through modifying source code, compromising build systems, or attacking the distribution of published software [13], [14]. Distribution attacks are mitigated by software producers signing binaries, so attacks on earlier stages of the pipeline are particularly attractive. Attacks on upstream software such as widely-utilized packages can affect multiple dependent products, potentially compromising whole ecosystems. Due to their multiparty nature, modeling and mitigating supply-chain threats requires considering both technical and social systems [15].

Open-source software represents a significant vector through which supply-chain attacks can be launched [16], and is ranked as one of OWASP's Top 10 web application security risks [17].

[1]According to scans by w3techs.com/technologies/details/en-utf8, 97% of the most accessed 10 million websites in 2021 use UTF-8 Unicode encodings.

III. ATTACK METHODOLOGY

A. *Reordering*

Internationalized text encodings require support for both left-to-right languages such as English and Russian, and right-to-left languages such as Hebrew and Arabic. When mixing scripts with different display orders, there must be a deterministic way to resolve conflicting directionality. For Unicode, this is implemented in the Bidirectional, or Bidi, Algorithm [3].

In some scenarios, the default ordering set by the Bidi Algorithm may not be sufficient; for these cases, override control characters are provided. Bidi overrides are invisible characters that enable switching the display ordering of groups of characters.

Table I provides a list of Bidi override characters relevant to this attack. Of note are `LRI` and `RLI`, which format subsequent text as left-to-right and right-to-left respectively, and are both closed by `PDI`.

Bidi overrides enable even single-script characters to be displayed in an order different from their logical encoding. This fact has previously been exploited to disguise the file extensions of malware disseminated by email [18] and to craft adversarial examples for NLP machine-learning pipelines [19].

As an example, consider the following Unicode character sequence:

```
RLI a b c PDI
```

which will be displayed as:

```
c b a
```

All Unicode Bidi overrides are restricted to affecting a single paragraph, as a newline character will implicitly close any overrides that lack a corresponding closing character.

B. *Isolate Shuffling*

In the Bidi specification, isolates are groups of characters that are treated as a single entity; that is, the entire isolate will be moved as a single block when the display order is overridden.

Isolates can be nested. For example, consider the Unicode character sequence:

```
RLI LRI a b c PDI LRI d e f PDI PDI
```

```
#include <stdio.h>
#include <string.h>

int main() {
    bool isAdmin = false;
    /*RLO } LRIif (isAdmin)PDI LRI begin admins only */
        printf("You are an admin.\n");
    /* end admin only RLO { LRI*/
    return 0;
}
```

Fig. 1. Encoded bytes of a Trojan-Source commenting-out attack in C.

```
#include <stdio.h>
#include <stdbool.h>

int main() {
    bool isAdmin = false;
    /* begin admins only */ if (isAdmin) {
        printf("You are an admin.\n");
    /* end admins only */ }
    return 0;
}
```

Fig. 2. Rendered text of a Trojan-Source commenting-out attack in C.

which will be displayed as:

```
d e f a b c
```

Embedding multiple layers of `LRI` and `RLI` within each other enables the near-arbitrary reordering of strings. This allows an adversary fine-grained control so they can manipulate the display order of text to diverge from its logically-encoded order.

C. Compiler Manipulation

Like most non-text rendering systems, compilers and interpreters do not typically process formatting control characters, including Bidi overrides, prior to parsing source code. This can be used to engineer a targeted gap between the visually-rendered source code as seen by a human eye, and the raw bytes of the encoded source code as evaluated by a compiler.

We can exploit this gap to create adversarially-encoded text that is understood differently by human reviewers and by compilers.

D. Syntax Adherence

Most well-designed programming languages will not allow arbitrary control characters in source code, as they will be viewed as tokens meant to affect the logic. Thus, randomly placing Bidi override characters in source code will typically result in a compiler or interpreter syntax error. To avoid such errors, we can exploit two general principles of programming languages:

- **Comments** – Most programming languages allow comments within which all text (including control characters) is ignored by compilers and interpreters.
- **Strings** – Most programming languages allow string literals that may contain arbitrary characters, including control characters.

While both comments and strings will have syntax-specific semantics indicating their start and end, these bounds are not respected by Bidi overrides. Therefore, by placing Bidi override characters exclusively within comments and strings, we can smuggle them into source code in a manner that most compilers will accept.

Making a random modification to the display order of characters on a line of valid source code is not particularly interesting, as it is very likely to be noticed by a human reviewer. The key insight is that as we can reorder source code characters in such a way that the resulting display order also represents syntactically valid source code.

E. Novel Supply-Chain Attack

Bringing all this together, we arrive at a novel supply-chain attack on source code. By injecting Unicode Bidi override characters into comments and strings, an adversary can produce syntactically-valid source code in most modern languages for which the display order of characters presents logic that diverges from the real logic, which is implemented by the encoded ordering of the same source-code characters. In effect, we anagram program A into program B.

Such an attack could be challenging for a human code reviewer to detect, as the rendered source code looks perfectly acceptable. If the change in logic is subtle enough to go undetected in subsequent testing, an adversary could introduce targeted vulnerabilities without being detected. We provide working examples of this attack in the following section.

Yet more concerning is the fact that Bidi override characters persist through the copy-and-paste functions on most modern browsers, editors, and operating systems. Any developer who copies code from an untrusted source into a protected code base may inadvertently introduce an invisible vulnerability.

F. Generality

We have implemented the above attack methodology, and the examples in the following section, with Unicode. Many modern compilers accept Unicode source code, as will be noted in our experimental evaluation. However, this attack paradigm should work with any text specification that enables the manipulation of display order, which is necessary to support internationalized text. Should the Unicode specification be supplanted by another standard, then in the absence of specific defenses, we believe that it is very likely to provide the same bidirectional functionality used to perform this attack.

IV. Exploit Techniques

There are a variety of ways to exploit the adversarial encoding of source code. The underlying principle is the same in each: use Bidi overrides to create a syntactically valid reordering of source code characters in the target language.

In the following section, we propose three general types of exploits that work across multiple languages. We do not claim that this list is exhaustive.

```
#!/usr/bin/env python3
bank = { 'alice': 100 }

def subtract_funds(account: str, amount: int):
    ''' Subtract funds from bank account then RLI''' ;return
    bank[account] -= amount
    return

subtract_funds('alice', 50)
```

Fig. 3. Encoded bytes of a Trojan-Source early-return attack in Python.

```
#!/usr/bin/env python3
bank = { 'alice': 100 }

def subtract_funds(account: str, amount: int):
    ''' Subtract funds from bank account then return; '''
    bank[account] -= amount
    return

subtract_funds('alice', 50)
```

Fig. 4. Rendered text of a Trojan-Source early-return attack in Python.

A. *Early Returns*

In the early-return exploit technique, adversaries disguise a genuine return statement as a comment or string literal, so they can cause a function to return earlier than it appears to.

Consider, for example, the case of docstrings – formal comments that purport to document the purpose of a function – which are considered good practice in software development. In languages where docstrings can be located within a function definition, an adversary need only find a plausible location to write the word `return` (or its language-specific equivalent) in a docstring comment, and then reorder the comment such that the `return` statement is executed immediately following the comment.

Figures 3 and 4 depict the encoded bytes and rendered text, respectively, of an early-return attack in Python3. Viewing the rendered text of the source code in fig. 4, one would expect the value of `bank['alice']` to be `50` after program execution. However, the value of `bank['alice']` remains `100` after the program executes. This is because the word `return` in the docstring is actually executed due to a Bidi override, causing the function to return prematurely and the code which subtracts value from a user's bank account to never run.

This technique is not specific to docstrings; any comment or string literal that can be manipulated by an adversary could hide an early-return statement.

B. *Commenting-Out*

In this exploit technique, text that appears to be legitimate code actually exists within a comment and is thus never executed. This allows an adversary to show a reviewer some code that appears to be executed but is not present from the perspective of the compiler or interpreter. For example, an adversary can comment out an important conditional, and then use Bidi overrides to make it appear to be still present.

This method is easiest to implement in languages that support mutliline comments. An adversary begins a line of code with a multiline comment that includes the code to be commented out and closes the comment on the same line. They then need only insert Bidi overrides to make it appear as if the comment is closed before the code via isolate shuffling.

Figures 1 and 2 depict the encoded bytes and rendered text, respectively, of a commenting-out attack in C. Viewing the rendered text makes it appear that, since the user is not an admin, no text should be printed. However, upon execution the program prints *You are an admin.*. The conditional doesn't actually exist; in the logical encoding, its text is wholly within the comment.

C. *Stretched Strings*

In this exploit technique, text that appears to be outside a string literal is actually located within it. This allows an adversary to manipulate string comparisons, for example causing strings which appear identical to give rise to a non-equal comparison.

Figures 5 and 6 depict the encoded bytes and rendered text, respectively, of a stretched-string attack in JavaScript. While it appears that the user's access level is "user" and therefore nothing should be written to the console, the code in fact outputs *You are an admin.* This is because the apparent comment following the comparison isn't actually a comment, but included in the comparison's string literal.

In general, the stretched-strings technique will allow an adversary to cause string comparisons to fail.

However, there are other, perhaps simpler, ways that an adversary can cause a string comparison to fail without visual effect. For example, the adversary can place invisible characters – that is, characters in Unicode that render to the absence of a glyph – such as the Zero Width Space[2] (ZWSP) into string literals used in comparisons. Although these invisible characters do not change the way a string literal renders, they will cause string comparisons to fail. Another option is to use characters that look the same, known as homoglyphs, such as the Cyrillic letter 'x' which typically renders identical to the Latin letter 'x' used in English but occupies a different code point. Depending on the context, the use of other character-encoding tricks may be more desirable than a stretched-string attack using Bidi overrides.

V. Evaluation

A. *Experimental Setup*

To validate the feasibility of the attacks described in this paper, we have implemented proof-of-concept attacks on simple programs in C, C++, C#, JavaScript, Java, Rust, Go, and Python. Each proof of concept is a program with source code that, when rendered, displays logic indicating that the program should have no output; however, the compiled version of each program outputs the text '*You are an admin.*' due to Trojan-Source attacks using Bidi override encodings.

[2]Unicode character `U+200B`

```
#!/usr/bin/env node

var accessLevel = "user";
if (accessLevel != "userRLO LRI// Check if adminPDI LRI") {
    console.log("You are an admin.");
}
```

Fig. 5. Encoded bytes of a Trojan-Source stretched-string attack in JavaScript.

```
#!/usr/bin/env node

var accessLevel = "user";
if (accessLevel != "user") { // Check if admin
    console.log("You are an admin.");
}
```

Fig. 6. Rendered text of a Trojan-Source stretched-string attack in JavaScript.

For this attack paradigm to work, the compilers or interpreters used must accept some form of Unicode input, such as UTF-8. We find that this is true for the overwhelming majority of languages in modern use. It is also necessary for the language to syntactically support modern internationalized text in string literals or comments.

Future compilers and interpreters may (and indeed should) employ defenses that emit errors or warnings when this attack is detected, but we have found no evidence of such behavior in any of our experiments.

All proofs of concept referenced in this paper have been made available online[3]. We have also created a website to help disseminate knowledge of this vulnerability pattern to all developer communities[4].

The following sections describe and evaluate Trojan-Source attack proofs-of-concept against specific programming languages.

B. C

In addition to supporting string literals, C supports both single-line and multi-line comments [20]. Single-line comments begin with the sequence // and are terminated by a newline character. Multi-line comments begin with the sequence /* and are terminated with the sequence */. Conveniently, multi-line comments can begin and end on a single line, despite their name. Strings literal are contained within double quotes, e.g. " · ". Strings can be compared using the function `strcmp`, which returns a falsey value when strings are equal, and a truthy value when strings are unequal.

As previously discussed, Figures 1 and 2 depict a commenting-out attack in C. We also provide an example of a Stretched-String attack in C in Appendix E Figures 24 and 25.

C is well-suited for the commenting-out and stretched-string exploit techniques, but only partially suited for early returns. This is because when the multiline comment terminator, i.e. `*/`, is reordered using a right-to-left override, it becomes `/*`. This provides a visual clue that something is not right. This can be overcome by writing reversible comment terminators as `/*/`, but this is less elegant and still leaves other visual clues such as the line-terminating semicolon. We provide an example of a functioning, but less elegant early-return attack in C in Appendix E Figures 26 and 27 which, although it looks like it prints '*Hello World.*' in fact prints nothing.

We have verified these attacks succeed on both GNU's `gcc` v7.5.0 (on Ubuntu) and Apple `clang` v12.0.5 (on MacOS).

[3]github.com/nickboucher/trojan-source
[4]trojansource.codes

C. C++

Since C++ is a linguistic derivative of C, it should be no surprise that the same attack paradigms work against the C++ specification [21]. Similar proof-of-concept programs modified to adhere to C++ preferred syntax can be seen in Appendix A Figures 8 to 11.

We have verified that both attacks succeed on GNU's `g++` v7.5.0 (on Ubuntu) and Apple `clang++` v12.0.5 (on MacOS).

D. C#

C# is an object-oriented language created by Microsoft that typically runs atop .NET, a cross-platform managed runtime, and is used heavily in corporate settings [22]. C# is vulnerable to the same attack paradigms as C and C++, and we present the same proof-of-concept attacks using C# syntax in Appendix B Figures 12 to 15.

We have verified that both attacks succeed on .NET 5.0 using the `dotnet-script` interpreter on MacOS.

E. JavaScript

JavaScript, also known as ECMAScript, is an interpreted language that provides in-browser client-side scripting for web pages, and is increasingly also used for server-side web application and API implementations [23]. JavaScript is vulnerable to the same attack paradigms as C, C++, and C#, and we present the same proof-of-concept attacks using JavaScript syntax in Appendix G Figures 32 and 33 as well as the previously discussed Figures 5 and 6.

We have verified that these attacks work against Node.js v16.4.1 (MacOS), which is a local JavaScript runtime built atop Chrome's V8 JavaScript Engine.

F. Java

Java is a bytecode-compiled multipurpose language maintained by Oracle [24]. It too is vulnerable to the same attack paradigms as C, C++, C#, and JavaScript, and we present the same proof-of-concept attacks using Java syntax in Appendix C Figures 16 to 19.

We have verified that these attacks work against OpenJDK v16.0.1 on MacOS.

G. Rust

Rust is a high-performance language increasingly used in systems programming [25]. It too is vulnerable to the same attack paradigms as C, C++, C#, JavaScript, and Java, and we present the same proof-of-concept attacks using Rust syntax in Appendix D Figures 20 to 23.

We have verified that these attacks work against Rust v1.53.0 (on MacOS), but note that one of the two proofs-of-concept (depicted in Figures 22 and 23) throws an unused variable warning on compilation.

H. Go

Go is a multipurpose open-source language produced by Google [26]. Go is vulnerable to the same attack paradigms as C, C++, C#, JavaScript, Java, and Rust, and we present the same proof-of-concept attacks using Go syntax in Appendix F Figures 28 and 29.

We have verified that these attacks work against Go v1.16.6 on MacOS. We note that unused variables throw compiler errors in the official Go compiler, and so our commenting-out Trojan-Source attack proof-of-concept deviates from our general pattern to ensure that no variables are left unused.

I. Python

Python is a general-purpose scripting language used heavily in data science and many other settings [27]. Python supports multiline comments in the form of docstrings opened and closed with `'''` or `"""`. We have already exploited this fact in Figures 3 and 4 to craft elegant early-return attacks.

An additional commenting-out proof-of-concept attack against Python 3 can be found in encoded form in Appendix H Figures 34 and 35.

We have verified that these attacks work against Python 3.9.5 compiled using `clang` 12.0.0 (on MacOS) and against Python 3.7.10) compiled using GNU's `gcc` (on Ubuntu).

VI. Discussion

A. Ethics

We have followed our department's ethical guidelines carefully throughout this research. We did not launch any attacks using Trojan-Source methods against codebases we did not own. Furthermore, we have performed responsible disclosure to all companies and organizations owning products in which we discovered vulnerabilities. At the time of submission, we are about halfway through a 90-day responsible-disclosure period, and we discuss some of our findings briefly later.

B. Attack Feasibility

Attacks on source code are both extremely attractive and highly valuable to motivated adversaries, as maliciously inserted backdoors can be incorporated into signed code that persists in the wild for long periods of time. Moreover, if backdoors are inserted into open source software components that are included downstream by many other applications, the blast radius of such an attack can be very large. Trojan-Source attacks introduce the possibility of inserting such vulnerabilities into source code invisibly, thus completely circumventing the current principal control against them, which is human source code review. This can make backdoors harder to detect and their insertion easier for adversaries to perform.

There is a long history of the attempted insertion of backdoors into critical code bases. One example was the attempted insertion of a root user escalation-of-privilege backdoor into the Unix kernel, which was as subtle as changing an == token to an = token [28]. This attack was detected by experienced developers seeing the vulnerability. The techniques described here allow a similar attack in the future to be invisible.

Recent research in developer security usability has documented that a significant portion of developers will gladly copy and paste insecure source code from unofficial online sources such as Stack Overflow[5] [29], [30]. Since Bidi overrides persist through standard copy-and-paste functionality, malicious code snippets with invisible vulnerabilities can be posted online in the hope that they will end up in production code. The market for such vulnerabilities is vibrant, with exploits on major platforms now commanding seven-figure sums [31].

Our experiments indicate that, as of the time of writing, C, C++, C#, JavaScript, Java, Rust, Go, and Python are all vulnerable to Trojan-Source attacks. More broadly, this class of attacks is likely applicable to any language with common compilers accepting Unicode source code input. Any entity whose security relies on the integrity of software supply chains should be concerned.

C. Syntax Highlighting

Many developers use text editors that, in addition to basic text editing features, provide syntax highlighting for the languages in which they are programming. Moreover, many code repository platforms, such as GitHub[6], provide syntax highlighting through a web browser. Comments are often displayed in a different color from code, and many of the proofs of concept provided in this paper work by deceiving developers into thinking that comments are code or vice versa.

We might have hoped that a well-implemented syntax-highlighting platform would at the very least exhibit unusual syntax highlighting in the vicinity of Bidi overrides in code, but our experience was mixed. Some attacks provided strange highlighting in a subset of editors, which may suffice to alert developers that an encoding issue is present. However, other attacks did not yield abnormal syntax highlighting in the same settings, and all syntax highlighting nuances were editor-specific.

Although unexpected coloring of source code may be enough to flag the possibility of an encoding attack to experienced developers, and in particular to those familiar with this work, we expect that most developers would not even notice unusual syntax highlighting, let alone investigate it thoroughly enough to work out what was going on. A motivated attacker could experiment with the visualization of different attacks in the text editors and code repository front-ends used in targeted organizations in order to select an attack that has no or minimal syntax highlighting effect on them.

Bidi overrides will typically cause a cursor to jump positions on a line when using arrow keys to click through tokens, or to highlight a line of text character-by-character. This is

[5]stackoverflow.com

[6]github.com

an artifact of the effect of the logical ordering of tokens on many operating systems and Unicode implementations. Such behavior, while producing no visible changes in text, may also be enough to alert some experienced developers. However, we suspect that this requires more attention than is given by most developers to reviews of large pieces of code.

D. Invisible Character Attacks

When discussing the string-stretching technique, we proposed that invisible characters or homoglyphs could be used to make visually-identical strings that are logically different when compared. Another invisible-vulnerability technique with with we experimented – largely without success – was the use of invisible characters in function names.

We theorized that invisible characters included in a function name could define a different function from the function defined by only the visible characters. This could allow an attacker to define an adversarial version of a standard function, such as `printf` in C, that can be invoked by calling the function with an invisible character in the function name. Such an adversarial function definition could be discretely added to a codebase by defining it in a common open-source package that is imported into the global namespace of the target program.

However, we found that all compilers analyzed in this paper emitted compilation errors when this technique was employed, with the exception of one compiler – Apple `clang` v12.0.5 – which emitted a warning instead of an error.

Should a compiler not instrument defenses against invisible characters in function definition names – or indeed in variable names – this attack may well be feasible. That said, our experimental evidence suggests that this theoretical attack already has defenses employed against it by most modern compilers, and thus is unlikely to succeed in practice.

E. Homoglyph Attacks

After we investigated invisible characters, we wondered whether homoglyphs in function names could be used to define distinct functions whose names appeared to the human eye to be the same. Then an adversary could write a function whose name appears the same as a pre-existing function – except that one letter is replaced with a visually similar character.

We were able to successfully implement homoglyph attack proofs-of-concept in every language discussed in this paper; that is, C, C++, C#, JavaScript, Java, Rust, Go, and Python all appear to be vulnerable. In our experiments, we defined two functions that appeared to have the name `sayHello`, except that one version used a Latin H while the other used a Cyrillic H.

Consider Figure 7, which implements a homoglyph attack in C++. For clarity, we denote the Latin H in blue and the Cyrillic H in red. This program outputs the text *Goodbye, World!* when compiled using `clang++`. Although this example program appears harmless, a homoglyph attack could cause significant damage when applied against a common function, perhaps via an imported library. For example, suppose a function called

```
#include <iostream>

void sayHello() {
    std::cout << "Hello, World!\n";
}

void sayНello() {
    std::cout << "Goodbye, World!\n";
}

int main() {
    sayНello();
    return 0;
}
```

Fig. 7. Homoglyph function attack in C++.

`hashPassword` was replaced with a similar function that called and returned the same value as the original function, but only after leaking the pre-hashed password over the network.

All compilers and interpreters examined in this paper emitted the text *Goodbye, World!* with similar proofs of concept. There were only three exceptions. GNU's `gcc` and its C++ counterpart, `g++`, both emitted stray token errors (although `clang` did not). Of particular note is the Rust compiler, which threw a 'mixed_script_confusables' warning while producing the homoglyph attack binary. The warning text suggested that the function name with the Cyrillic H used "mixed script confusables" and suggested rechecking to ensure usage of the function was wanted. This is a well-designed defense against homoglyph attacks, and it indicates that this attack has been seriously considered by at least one compiler team. We wish that the other compilers examined also exhibited this behavior.

This defense, together with the defenses against invisible character attacks, should serve as a precedent. It is reasonable to expect compilers to also incorporate defences against Trojan-Source attacks. But as we will now discuss, some compiler maintainers have resisted this; and there are other defence options too.

F. Defenses

The simplest defense is to ban the use of text directionality control characters both in language specifications and in compilers implementing these languages.

In most settings, this simple solution may well be sufficient. If an application wishes to print text that requires Bidi overrides, developers can generate those characters using escape sequences rather than embedding potentially dangerous characters into source code.

This simple defense can be improved by adding a small amount of nuance. By banning all directionality-control characters, users with legitimate Bidi-override use cases in comments are penalized. Therefore, a better defense might be to ban the use of *unterminated* Bidi override characters within string literals and comments. By ensuring that each override is terminated – that is, for example, that every `LRI` has a matching `PDI` – it becomes impossible to distort legitimate source code outside of string literals and comments.

Trojan-Source defenses must be enabled by default on all compilers that support Unicode input, and turning off the defenses should only be permitted when a dedicated suppression flag is passed.

While changes to language specifications and compilers are ideal solutions, there is an immediate need for existing code bases to be protected against this family of attacks. Moreover, some languages or compilers may choose not to implement appropriate defenses. To protect organizations that rely on them, defenses can be employed in build pipelines, code repositories, and text editors.

Build pipelines, such as those used by software producers to build and sign production code, can scan for the presence of Bidi overrides before initiating each build and break the build if such a character is found in source code. Alternatively, build pipelines can scan for the more nuanced set of unterminated Bidi overrides. Such tactics provide an immediate and robust defense for existing software maintainers.

Code repository systems and text editors can also help prevent Trojan-Source attacks by making them visible to human reviewers. For example, code repository front-ends, such as web UIs for viewing committed code, can choose to represent Bidi overrides as visible tokens, thus making attacks visible, and by adding a visual warning to the affected lines of code.

Code editors can employ similar tactics. In fact, some already do; `vim`, for example, defaults to showing Bidi overrides as numerical code points rather than applying the Bidi algorithm. However, many common code editors do not adopt this behavior, including most GUI editors such as, at the time of writing, Microsoft's VS Code and Apple's Xcode.

G. Responsible Disclosure

One issue we faced when performing responsible disclosure for Trojan-Source attacks was the tragedy of the commons: because this attack affects such a broad ecosystem, and because it can be defended from multiple different layers, we often received the response that the issue seemed to be someone else's problem.

Compiler maintainers often suggested that the problem should be tackled by code editors, while maintainers of code editors in their turn suggested that the Trojan-Source vulnerability was a bug in compilers. Some online code repositories stated that it was outside of their threat models.

Another pattern we observed concerned the use of outsourced vulnerability disclosure platforms where security researchers are invited to submit for bug bounties. Our experience with these platforms was that, while they were reasonably designed for the disclosure of well-established vulnerability patterns such as SQL injection and cross-site-scripting, they were not well suited for the disclosure of newer methods that did not fit existing patterns. On many cases, these platforms labelled Trojan-Source attacks as posing no threat. In some of these cases, we learned that vulnerabilities reported through outsourced platforms do not go to the security teams of the recipient companies, but rather are handled by the outsourced third party. When we contacted the security teams of the companies whose outsourced security platforms had rejected our disclosure, we found that team members were interested in the attack and willing to engage.

Our experience so far is two-fold. First, the industry as a whole struggles with coordinated disclosure when a large ecosystem is involved; and second, that companies which outsource security work may save costs, but introduce new and subtle failure modes, which may be worthy of more systematic study.

VII. Conclusion

We have presented a new type of attack that enables invisible vulnerabilities to be inserted into source code. Our Trojan-Source attack uses Unicode control characters to modify the order in which blocks of characters are displayed, thus enabling comments to appear to be code and vice versa. This enables an attacker to craft code that is interpreted one way by compilers and a different way by human reviewers. We present proofs of concept for C, C++, C#, JavaScript, Java, Rust, Go, and Python, and argue that this attack may well appear in any programming language that supports internationalized text in comments and string literals, even in other encoding standards.

As powerful supply-chain attacks can be launched easily using these techniques, it is essential for organizations that participate in a software supply chain to implement defenses. We have discussed countermeasures that can be used at a variety of levels in the software development toolchain: the language specification, the compiler, the text editor, the code repository, and the build pipeline. We are of the view that the long-term solution to the problem will be deployed in compilers. We note that almost all compilers already defend against one related attack, which involves creating adversarial function names using zero-width space characters, while three generate errors in response to another, which involves using homoglyphs in function names.

However, as some compiler maintainers are dragging their feet, it is prudent to deploy other controls in the meantime where this is quick and cheap, or relevant and needful. We also recommend that governments and firms that rely on critical software should exert pressure on their suppliers to ensure that they implement adequate defences.

Scientifically, this paper also contributes to the growing body of work on security usability from the developer's perspective. It is not sufficient for a compiler to be verified; it must also be safely usable. Compilers that are trivially vulnerable to adversarial text encoding cannot reasonably be described as secure.

References

[1] K. Thompson, "Reflections on trusting trust," *Commun. ACM*, vol. 27, no. 8, pp. 761–763, 1984. [Online]. Available: https://doi.org/10.1145/358198.358210

[2] S. Peisert, B. Schneier, H. Okhravi, F. Massacci, T. Benzel, C. Landwehr, M. Mannan, J. Mirkovic, A. Prakash, and J. Michael, "Perspectives on the solarwinds incident," *IEEE Security & Privacy*, vol. 19, no. 02, pp. 7–13, mar 2021.

[3] The Unicode Consortium, "Unicode Bidirectional Algorithm," The Unicode Consortium, Tech. Rep. Unicode Technical Report #9, Feb. 2020. [Online]. Available: https://www.unicode.org/reports/tr9/tr9-42.html

[4] J. Painter and J. McCarthy, "Correctness of a compiler for arithmetic expressions," in *Proceedings of Symposia in Applied Mathematics*, vol. 19. American Mathematical Society, 1967, pp. 33–41. [Online]. Available: http://jmc.stanford.edu/articles/mcpain/mcpain.pdf

[5] M. A. Dave, "Compiler verification: a bibliography," *ACM SIGSOFT Software Engineering Notes*, vol. 28, no. 6, pp. 2–2, 2003.

[6] D. Patterson and A. Ahmed, "The next 700 compiler correctness theorems (functional pearl)," *Proceedings of the ACM on Programming Languages*, vol. 3, no. ICFP, pp. 1–29, 2019.

[7] V. D'Silva, M. Payer, and D. Song, "The correctness-security gap in compiler optimization," in *2015 IEEE Security and Privacy Workshops*, 2015, pp. 73–87.

[8] L. Simon, D. Chisnall, and R. Anderson, "What you get is what you C: Controlling side effects in mainstream C compilers," in *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, Apr. 2018, pp. 1–15.

[9] The Unicode Consortium, "The Unicode Standard, Version 13.0," Mar. 2020. [Online]. Available: https://www.unicode.org/versions/Unicode13.0.0

[10] C. J. Alberts, A. J. Dorofee, R. Creel, R. J. Ellison, and C. Woody, "A systemic approach for assessing software supply-chain risk," in *2011 44th Hawaii International Conference on System Sciences*, 2011, pp. 1–8.

[11] A. Nappa, R. Johnson, L. Bilge, J. Caballero, and T. Dumitras, "The attack of the clones: A study of the impact of shared code on vulnerability patching," in *2015 IEEE Symposium on Security and Privacy*, 2015, pp. 692–708.

[12] J. Biden, "Executive Order on Improving the Nation's Cybersecurity," May 2021, Executive Order 14028. [Online]. Available: https://www.whitehouse.gov/briefing-room/presidential-actions/2021/05/12/executive-order-on-improving-the-nations-cybersecurity

[13] R. J. Ellison and C. Woody, "Supply-chain risk management: Incorporating security into software development," in *2010 43rd Hawaii International Conference on System Sciences*, 2010, pp. 1–10.

[14] E. Levy, "Poisoning the software supply chain," *IEEE Security Privacy*, vol. 1, no. 3, pp. 70–73, 2003.

[15] B. A. Sabbagh and S. Kowalski, "A socio-technical framework for threat modeling a software supply chain," *IEEE Security Privacy*, vol. 13, no. 4, pp. 30–39, 2015.

[16] M. Ohm, H. Plate, A. Sykosch, and M. Meier, "Backstabber's knife collection: A review of open source software supply chain attacks," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, C. Maurice, L. Bilge, G. Stringhini, and N. Neves, Eds. Cham: Springer International Publishing, 2020, pp. 23–43.

[17] OWASP, "A9:2017 Using Components with Known Vulnerabilities," 2017. [Online]. Available: https://owasp.org/www-project-top-ten/2017/A9_2017-Using_Components_with_Known_Vulnerabilities.html

[18] Brian Krebs, "'Right-to-Left Override' Aids Email Attacks," Sep. 2011. [Online]. Available: https://krebsonsecurity.com/2011/09/right-to-left-override-aids-email-attacks/

[19] N. Boucher, I. Shumailov, R. Anderson, and N. Papernot, "Bad Characters: Imperceptible NLP Attacks," 2021.

[20] ISO, *ISO/IEC 9899:2018 Information technology — Programming languages — C*, 4th ed. Geneva, Switzerland: International Organization for Standardization, Jun. 2018. [Online]. Available: https://www.iso.org/standard/74528.html

[21] ISO, *ISO/IEC 14882:2020 Information technology — Programming languages — C++*, 6th ed. Geneva, Switzerland: International Organization for Standardization, Dec. 2020. [Online]. Available: https://www.iso.org/standard/79358.html

[22] ISO, *ISO/IEC 23270:2018 Information technology — Programming languages — C#*, 3rd ed. Geneva, Switzerland: International Organization for Standardization, Dec. 2018. [Online]. Available: https://www.iso.org/standard/75178.html

[23] Ecma, *ECMA-262*, 12th ed. Geneva, Switzerland: Ecma International, Jun. 2021. [Online]. Available: https://www.ecma-international.org/publications-and-standards/standards/ecma-262

[24] J. Gosling, B. Joy, G. Steele, G. Bracha, A. Buckley, D. Smith, and G. Bierman, *The Java® Language Specification*, 16th ed. Java Community Press, Feb. 2021. [Online]. Available: https://docs.oracle.com/javase/specs/jls/se16/jls16.pdf

[25] The Rust Project Developers, *The Rust Reference*. The Rust Foundation, 2018. [Online]. Available: https://doc.rust-lang.org/reference

[26] The Go Project Developers, *The Go Programming Language Specification*. Google, Feb. 2021. [Online]. Available: https://golang.org/ref/spec

[27] The Python Project Developers, *The Python Language Reference*, 3rd ed. The Python Software Foundation, 2018. [Online]. Available: https://docs.python.org/3/reference

[28] J. Corbet, "An attempt to backdoor the kernel," *Linux Weekly News*, Nov. 2003. [Online]. Available: https://lwn.net/Articles/57135

[29] Y. Acar, M. Backes, S. Fahl, D. Kim, M. L. Mazurek, and C. Stransky, "You get where you're looking for: The impact of information sources on code security," in *2016 IEEE Symposium on Security and Privacy (SP)*, 2016, pp. 289–305.

[30] F. Fischer, K. Böttinger, H. Xiao, C. Stransky, Y. Acar, M. Backes, and S. Fahl, "Stack overflow considered harmful? the impact of copy amp;paste on android application security," in *2017 IEEE Symposium on Security and Privacy (SP)*, 2017, pp. 121–136.

[31] N. Perlroth, *This Is How They Tell Me the World Ends: The Cyberweapons Arms Race*. Bloomsbury, 2021.

Appendix

A. *C++ Trojan-Source Proofs-of-Concept*

```
#include <iostream>
#include <string>

int main() {
    std::string access_level = "user";
    if (access_level.compare("userRLO LRI// Check if adminPDI LRI")) {
        std:cout << "You are an admin.\n";
    }
    return 0;
}
```

Fig. 8. Encoded bytes of a Trojan-Source stretched-string attack in C++.

```
#include <iostream>
#include <string>

int main() {
    std::string access_level = "user";
    if (access_level.compare("user")) { // Check if admin
        std::cout << "You are an admin.\n";
    }
    return 0;
}
```

Fig. 9. Rendered text of a Trojan-Source stretched-string attack in C++.

```
#include <iostream>

int main() {
    bool isAdmin = false;
    /*RLO } LRIif (isAdmin)PDI LRI begin admins only */
        std::cout << "You are an admin.\n";
    /* end admin only RLO { LRI*/
    return 0;
}
```

Fig. 10. Encoded bytes of a Trojan-Source commenting-out attack in C++.

```
#include <iostream>

int main() {
    bool isAdmin = false;
    /* begin admins only */ if (isAdmin) {
        std::cout << "You are an admin.\n";
    /* end admins only */ }
    return 0;
}
```

Fig. 11. Rendered text of a Trojan-Source commenting-out attack in C++.

B. *C# Trojan-Source Proofs-of-Concept*

```
#!/usr/bin/env dotnet-script

string access_level = "user";
if (access_level != "userRLO LRI// Check if adminPDI LRI") {
    Console.WriteLine("You are an admin.");
}
```

Fig. 12. Encoded bytes of a Trojan-Source stretched-string attack in C#.

```
#!/usr/bin/env dotnet-script

string access_level = "user";
if (access_level != "user") { // Check if admin
    Console.WriteLine("You are an admin.");
}
```

Fig. 13. Rendered text of a Trojan-Source stretched-string attack in C#.

```
#!/usr/bin/env dotnet-script

bool isAdmin = false;
/*RLO } LRIif (isAdmin)PDI LRI begin admins only */
    Console.WriteLine("You are an admin.");
/* end admin only RLO { LRI*/
```

Fig. 14. Encoded bytes of a Trojan-Source commenting-out attack in C#.

```
#!/usr/bin/env dotnet-script

bool isAdmin = false;
/* begin admins only */ if (isAdmin) {
    Console.WriteLine("You are an admin.");
/* end admins only */ }
```

Fig. 15. Rendered text of a Trojan-Source commenting-out attack in C#.

C. *Java Trojan-Source Proofs-of-Concept*

```
public class TrojanSource {
    public static void main(String[] args) {
        String accessLevel = "user";
        if (accessLevel != "userRLO LRI// Check if adminPDI LRI") {
            System.out.println("You are an admin.");
        /* end admin only RLO { LRI*/
    }
}
```

Fig. 16. Encoded bytes of a Trojan-Source stretched-string attack in Java.

```
public class TrojanSource {
    public static void main(String[] args) {
        String accessLevel = "user";
        if (accessLevel != "user") { // Check if admin
            System.out.println("You are an admin.");
        }
    }
}
```

Fig. 17. Rendered text of a Trojan-Source stretched-string attack in Java.

```
public class TrojanSource {
    public static void main(String[] args) {
        boolean isAdmin = false;
        /*RLO } LRIif (isAdmin)PDI LRI begin admins only */
            System.out.println("You are an admin.");
        /* end admin only RLO { LRI*/
    }
}
```

Fig. 18. Encoded bytes of a Trojan-Source commenting-out attack in Java.

```
public class TrojanSource {
    public static void main(String[] args) {
        boolean isAdmin = false;
        /* begin admins only */ if (isAdmin) {
            System.out.println("You are an admin.");
        /* end admins only */ }
    }
}
```

Fig. 19. Rendered text of a Trojan-Source commenting-out attack in Java.

D. *Rust Trojan-Source Proofs-of-Concept*

```
fn main() {
    let access_level = "user";
    if (access_level != "userRLO LRI// Check if adminPDI LRI") {
        println!("You are an admin.");
    }
}
```

Fig. 20. Encoded bytes of a Trojan-Source stretched-string attack in Rust.

```
fn main() {
    let access_level = "user";
    if access_level != "user" { // Check if admin
        println!("You are an admin.");
    }
}
```

Fig. 21. Rendered text of a Trojan-Source stretched-string attack in Rust.

```
fn main() {
    let is_admin = false;
    /*RLO } LRIif is_adminPDI LRI begin admins only */
        println!("You are an admin.");
    /* end admin only RLO { LRI*/
}
```

Fig. 22. Encoded bytes of a Trojan-Source commenting-out attack in Rust.

```
fn main() {
    let is_admin = false;
    /* begin admins only */ if is_admin {
        println!("You are an admin.");
    /* end admins only */ }
}
```

Fig. 23. Rendered text of a Trojan-Source commenting-out attack in Rust.

E. C Trojan-Source Proofs-of-Concept

```
#include <stdio.h>
#include <string.h>

int main() {
    char* access_level = "user";
    if (strcmp(access_level, "userRLO LRI// Check if adminPDI LRI")) {
        printf("You are an admin.\n");
    }
    return 0;
}
```

Fig. 24. Encoded bytes of a Trojan-Source stretched-string attack in C.

```
#include <stdio.h>
#include <string.h>

int main() {
    char* access_level = "user";
    if (strcmp(access_level, "user")) { // Check if admin
        printf("You are an admin.\n");
    }
    return 0;
}
```

Fig. 25. Rendered text of a Trojan-Source stretched-string attack in C.

```
#include <stdio.h>

int main() {
    /* Say hello; newline RLI /*/ return 0 ;
    printf("Hello world.\n");
    return 0;
}
```

Fig. 26. Encoded bytes of a Trojan-Source early-return attack in C.

```
#include <stdio.h>

int main() {
    /* Say hello; newline; return 0 /*/
    printf("Hello world.\n");
    return 0;
}
```

Fig. 27. Rendered text of a Trojan-Source early-return attack in C.

F. Go Trojan-Source Proofs-of-Concept

```
package main

import "fmt"

func main() {
    var accessLevel = "user"
    if access_level != "userRLO LRI// Check if adminPDI LRI" {
        fmt.Println("You are an admin.")
    }
}
```

Fig. 28. Encoded bytes of a Trojan-Source stretched-string attack in Go.

```
package main

import "fmt"

func main() {
        var accessLevel = "user"
        if accessLevel != "user" { // Check if admin
                fmt.Println("You are an admin.")
        }
}
```

Fig. 29. Rendered text of a Trojan-Source stretched-string attack in Go.

```
package main

import "fmt"

func main() {
    var isAdmin = false
    var isSuperAdmin = false
    isAdmin = isAdmin || isSuperAdmin
    /*RLO } LRIif (isAdmin)PDI LRI begin admins only */
        fmt.Println("You are an admin.")
    /* end admin only RLO { LRI*/
}
```

Fig. 30. Encoded bytes of a Trojan-Source commenting-out attack in Go.

```
package main

import "fmt"

func main() {
    var isAdmin = false
    var isSuperAdmin = false
    isAdmin = isAdmin || isSuperAdmin
    /* begin admins only */ if (isAdmin) {
        fmt.Println("You are an admin.")
    /* end admins only */ }
}
```

Fig. 31. Rendered text of a Trojan-Source commenting-out attack in Go.

G. *JavaScript Trojan-Source Proof-of-Concept*

```
#!/usr/bin/env node

var isAdmin = false;
/*RLO } LRIif (isAdmin)PDI LRI begin admins only */
    console.log("You are an admin.");
/* end admin only RLO { LRI*/
```

Fig. 32. Encoded bytes of a Trojan-Source commenting-out attack in JS.

```
#!/usr/bin/env node

var isAdmin = false;
/* begin admins only */ if (isAdmin) {
    console.log("You are an admin.");
/* end admins only */ }
```

Fig. 33. Rendered text of a Trojan-Source commenting-out attack in JS.

H. *Python Trojan-Source Proof-of-Concept*

```
#!/usr/bin/env python3

access_level = "user"
if access_level != 'noneRLOLRI': # Check if admin PDILRI' and access_level != 'user
    print("You are an admin.\n");
```

Fig. 34. Encoded bytes of a Trojan-Source commenting-out attack in Python.

```
#!/usr/bin/env python3

access_level = "user"
if access_level != 'none' and access_level != 'user': # Check if admin
    print("You are an admin.")
```

Fig. 35. Rendered text of a Trojan-Source commenting-out attack in Python.