1st presentation

Trojan Source: Invisible Vulnerabilities

Research Paper

Group 1

Lin Pengcheng – 1008527 **Baskar Durga** – 1008534 **Harish Navnit** – 1008538 **Tang Fei** - 1008558

CONTENTS

Trojan Source: Invisible Vulnerabilities

Introduction PART 01

This paper is included in the Proceedings of the 32nd USENIX Security Symposium.

01

Souce: https://www.usenix.org/conference/usenixsecurity23/presentation/boucher

Trojan Source:
Invisible Vulnerabilities

Nicholas Boucher
University of Cambridge
Computer Science & Technology
nicholas.boucher@cl.cam.ac.uk

Ross Anderson
University of Cambridge
and 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 of supply-chain compromise across the industry. We present working examples of Trojan Source attacks in C, C++, C#, JavaScript, Java, Rust, Go, Python, SQL, Bash, Assembly, and Solidity. 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. We document an industry-wide coordinated disclosure for these vulnerabilities; as they affect most compilers, editors, and repositories, the exercise teaches how different firms, open-source communities, and other stakeholders respond to vulnerability disclosure.

1 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 correctly implementing 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 a supply-chain attacker who seeks to inject vulnerabilities into software upstream of the ultimate targets, as happened in 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, 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 now demonstrate.

In this paper, we make the following contributions:

- We define a novel class of vulnerabilities, which we call Trojan Source attacks, and which use maliciously encoded but 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, Python, SQL, Bash, Assembly, and Solidity.
- We describe effective defenses that must be employed by compilers, as well as other defenses that can be used in editors, repositories, and build pipelines, and discuss the limitations of these defenses.
- We document the coordinated disclosure process we used to disclose this vulnerability across the industry, and what it teaches about the response to disclosure.
- We raise a new question about what it means for a compiler to be trustworthy.

2 Background

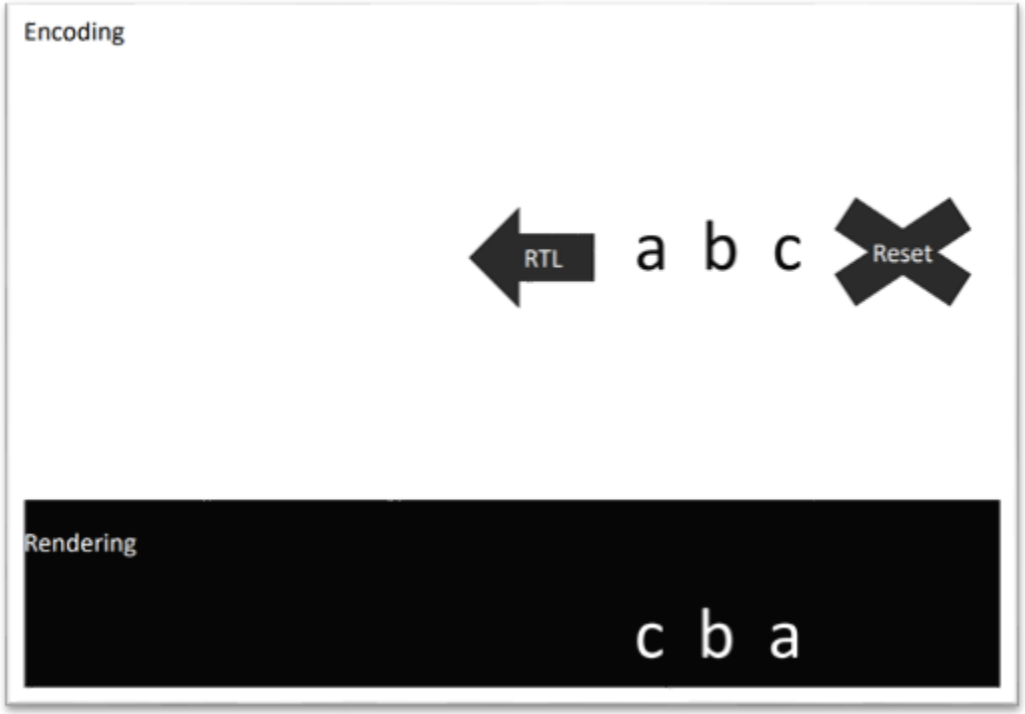
2.1 Compiler Security

Compilers translate high-level programming languages into lower-level representations such as architecture-specific machine instructions or portable bytecode. They seek to im-

What is the Trojan Source Attack?

- **Concealed Malware**: Malicious code hidden within legitimate software.
- **Hidden Intentions**: Malware's true purpose is disguised to evade detection.
- **Exploitation of Trust**: Users are misled into installing infected software due to its apparent legitimacy.
- **Backdoor Access**: Trojan Source enables unauthorized access to systems.
- **Data Theft or Manipulation**: Used for stealing sensitive information or altering data.

Encoding and Rendering



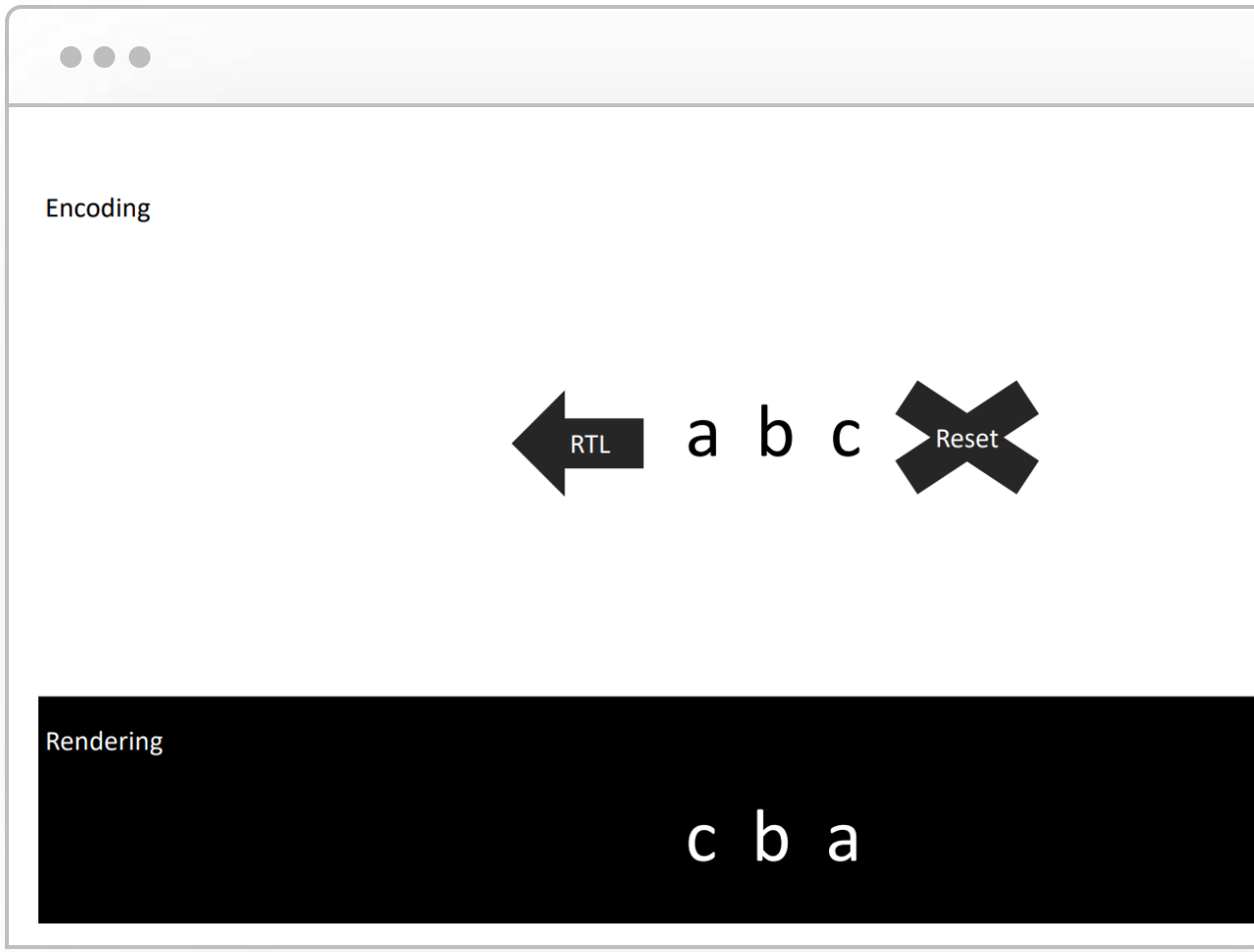
Example

1. Encoding (How the code is written and stored):

Right-to-Left Override (RTL): This is a special control character used in bidirectional text to indicate that the following text should be read from right to left. It is typically used for languages like Arabic or Hebrew but can be misused in an attack.
The Characters 'a b c': These are the visible characters that are placed between the RTL control character and the reset control character.
Reset: This is another control character that resets the directionality of the text, in this case back to left-to-right, which is the standard for most programming languages.

2. Rendering (How the code is displayed):

'c b a': Because of the RTL control character, the text editor renders the characters in reverse order. This can make code appear in a different sequence than it would be executed, potentially hiding malicious code within seemingly benign code when viewed.



The Vulnerability

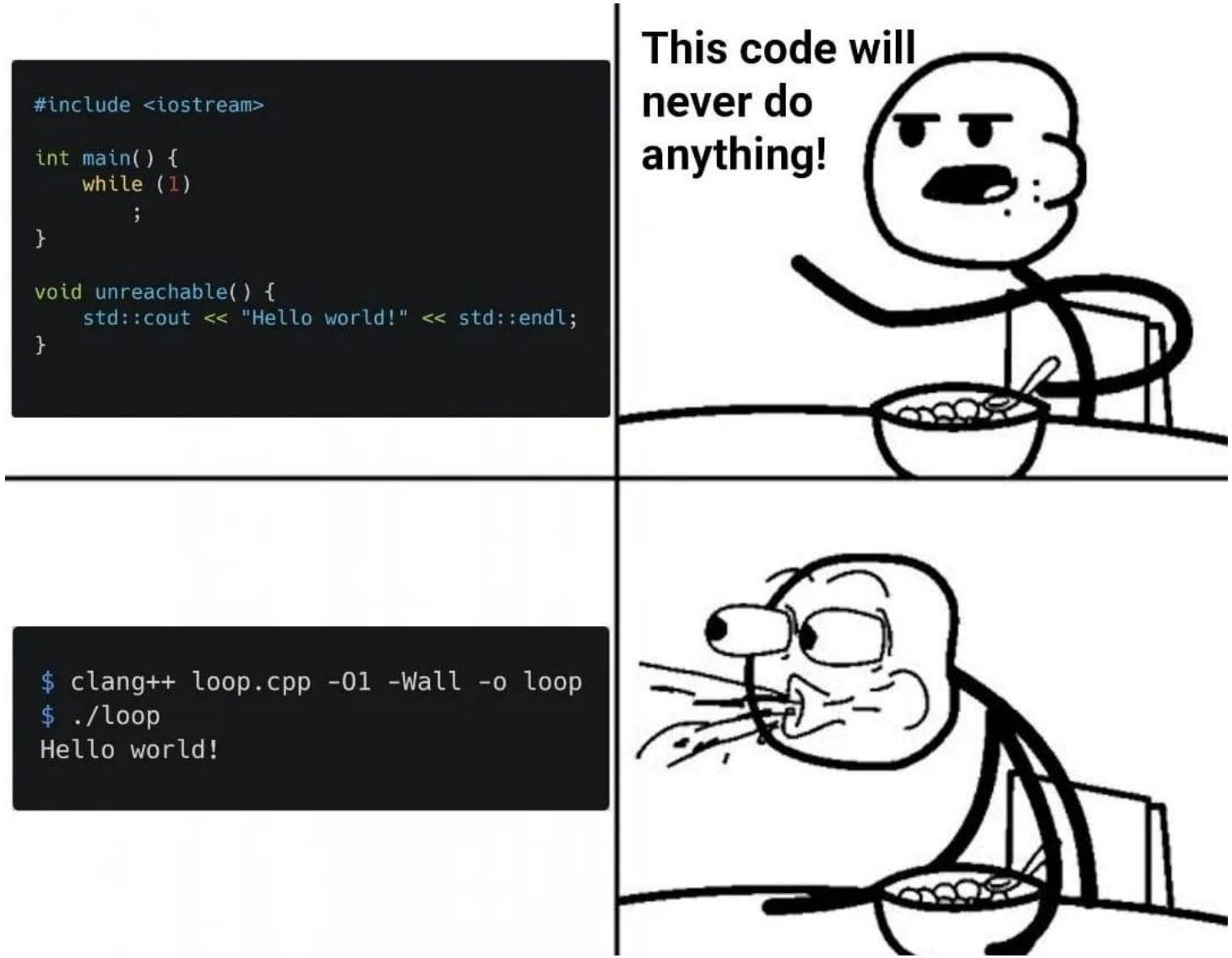
- 1. Control characters can override text direction.
- 2. This can modify the display order.
- 3. They can be placed into comments and strings.

⇒ **Evil program A to be anagrammed into benign program B**

Problem Background

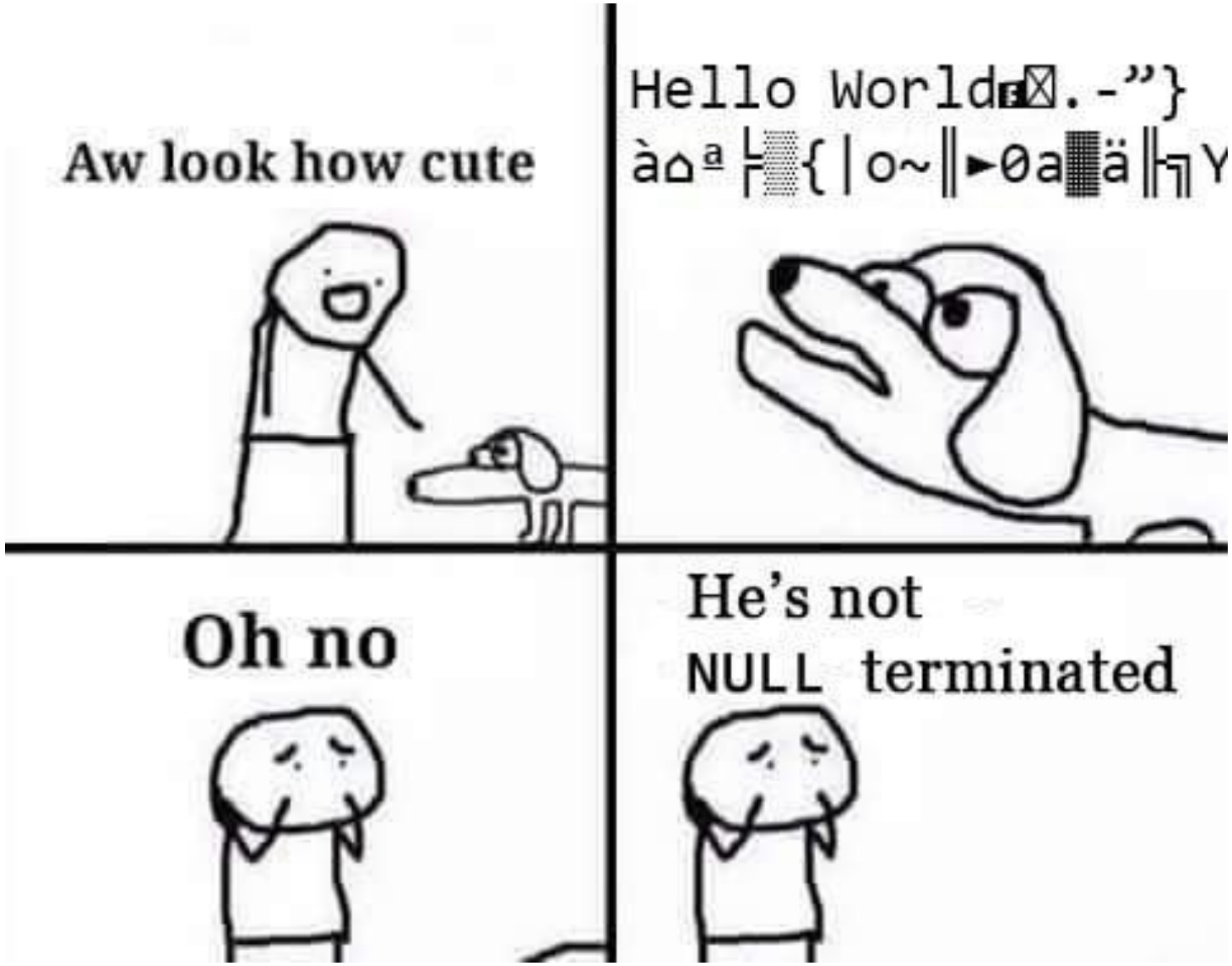
Compilers

- Compilers translate high-level programming languages into lower-level machine instructions.
- Compilers introduce several optimizations to carry out the above translation.
- Many a times there are discrepancies between source code logic & compiler output logic.



Text Encodings

- Modern text encodings have standardized around Unicode specification.
- Unicode defines 143,859 different characters, including control characters.
- Text rendering is performed by interpreting encoded bytes and finally displaying glyphs provided for each character in the chosen font.



By injecting unicode Bidi control characters into comments and strings, an adversary can produce syntactically valid source code.

```
#!/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)
```

Figure 1: Encoded bytes of a Trojan Source early-return attack in Python.

https://gist.github.com/tinvaan/43fce85bbd8e5e368380513a2fede896

By injecting unicode Bidi control characters into comments and strings, an adversary can produce syntactically valid source code.

```
#!/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)
```

Figure 2: Rendered text of a Trojan Source early-return attack in Python.

https://gist.github.com/tinvaan/43fce85bbd8e5e368380513a2fede896


Security Problem ?

Supply chain attacks

POTUS' executive order on improving nation's cybersecurity.

Supply chain attacks

An adversary attempts to exploit vulnerabilities in the software that is being used to serve the target software.

Open source packages constitute the most prominent supply chain attack vectors.

Backdoors in open source components maybe included downstream by other applications, thus making the blast radius of such attacks unintendedly large.

Trojan source attacks introduce the possibility of inserting such vulnerabilities into source code invisibly, thus circumventing the current principal control against them – Human source code reviews.

Attack and Defense Mechanisms

Attack Mechanisms

Encoding Manipulation	• Manipulating source code encoding to introduce hidden vulnerabilities.
Visual Discrepancies	• Making it challenging to detect vulnerabilities through traditional code reviews.
Introduction of Invisible Vulnerabilities	• Vulnerabilities not easily perceptible by human reviewers or automated tools.

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 control characters are restricted to affecting a single paragraph, as a newline character will explicitly close any unbalanced control characters, namely those that lack a corresponding closing character.

3.2 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

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 gives an adversary fine-grained control, so they can manipulate the display order of text into an anagram of its logically-encoded order.

3.3 Compiler Manipulation

Like most non-text rendering systems, compilers and interpreters do not typically process formatting control characters, including Bidi control characters, 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.

3.4 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 control 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 control characters. Therefore, by placing Bidi control 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. Our key insight is that we can reorder source code characters in such a way that the resulting display order also represents syntactically valid source code.

3.5 Novel Supply-Chain Attack

Bringing all this together, we arrive at a novel supply-chain attack on source code. By injecting Unicode Bidi control 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. 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 control 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. Code copying is already a significant source of real-world security exploits [20].

3.6 Threat Model

More formally, we define the threat model for Trojan Source attacks as an active adversary who seeks to inject adversarial logic into targeted software. If such software has an upstream dependency on further software, the adversary may target that instead in a supply chain attack. We define the adversary as having the following access:

Defense Mechanisms

Compiler-Level Defenses

- Definitive Compiler Checks
- Enhanced Compilation Process

Additional Mitigating Controls

- Code Analysis Tools
- Repository Scanning
- Build Pipeline Security

Collaboration and Coordinated Disclosure

- Coordinated Vulnerability Disclosure
- Collective Response

7.6 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 control characters, 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 control character use cases in comments are penalized. Therefore, a better defense might be to ban the use of *unterminated* Bidi control characters within string literals and comments. By ensuring that each control character 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 control characters 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 control characters. 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 control characters 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 control characters as numerical code points rather than applying the Bidi algorithm. However, many common code editors did not adopt this behavior at the time of disclosure, including most GUI editors such as Microsoft's VS Code and Apple's Xcode.

Many of the largest compilers, code editors, and repositories adopted these defenses following a coordinated disclosure process; we will describe more detail, including caveats about false positives, later in this section.

PART 04

Conclusion

• In conclusion, this article proposed a new type of attack that exploits the subtleties of text encoding standards to maliciously encode source code, making it appear differently to compilers and human eyes, and proposes compiler-level defenses to prevent such attacks.

PART 04

Execution Plan

- Analysis Phase:
  1. Understand the principle of Trojan Source attacks and how does it work.
  2. Collect actual cases or historical data to show the application and consequences of Trojan Source attacks in the real world.
- Proposal Phase:
  1. Summarize the main characteristics, harms, and scope of influence of Trojan Source.
  2. Design or improve compilers to detect and reject Trojan Source.
  3. Propose other possible defense measures.
- Verification Phase:
  1. Simulate experiments to verify the effectiveness of the scheme.
  2. Effect evaluation. Use charting tools to display data and results.
  3. Summarize the advantages and disadvantages of the scheme and propose possible improvements and optimization suggestions.