Improving Malware Detection By Parsing Broken Code

Abhay Garg
Alexandre Ginet
Arijit De
Paneendra BA
Yu Zhou

Follow this and additional works at: https://www.tdcommons.org/dpubs_series

Recommended Citation
Garg, Abhay; Ginet, Alexandre; De, Arijit; BA, Paneendra; and Zhou, Yu, "Improving Malware Detection By Parsing Broken Code", Technical Disclosure Commons, (November 24, 2020)
https://www.tdcommons.org/dpubs_series/3799

This work is licensed under a Creative Commons Attribution 4.0 License.
This Article is brought to you for free and open access by Technical Disclosure Commons. It has been accepted for inclusion in Defensive Publications Series by an authorized administrator of Technical Disclosure Commons.
Improving Malware Detection By Parsing Broken Code

ABSTRACT

Spreadsheets, word processors, and other document editing applications enable users to write scripts or macros that automate a sequence of actions, e.g., keystrokes, mouse-clicks, etc. through code. Although macros can improve user efficiency by automating repetitive actions, executable code within a document can also potentially include malware. Macro-based malware is known to intentionally use broken syntax to bypass detection. This disclosure describes a parser that is resilient to syntax errors in code, and which can, by applying local corrections, continue to parse the rest of the code after encountering a parse error. Once corrected, the code can be subject to malware detection prior to or after translation into the target language.

KEYWORDS

- Parse error
- Syntax error
- Broken code
- Partial runnable code
- Abstract syntax tree
- Indicator of Compromise (IoC)
- Static analysis
- Malware detection
- Spreadsheet macro
BACKGROUND

Spreadsheets, word processors, and other document editing applications enable users to write scripts or macros that automate a sequence of actions, e.g., keystrokes, mouse-clicks, etc. through code. Although macros can improve user efficiency by automating repetitive actions, executable code within a document can also potentially include malware. Macro-based malware is known to intentionally use broken syntax to bypass detection. Some examples of broken syntax are if, while, or for blocks that don’t have an end-statement; premature ending of a function; etc.

Spreadsheet or other applications typically execute a macro until a syntax error is encountered. A syntax error in one function does not generally affect other functions. However, syntax errors can impair or stall the translation (or compilation) of a macro when translating into another language. Furthermore, broken code cannot be tested robustly for the presence of malware. Some Example language translations include translation from VBA to JavaScript, from Perl to Python, etc.

Although there are language parsers available that can parse a valid program and generate an abstract syntax tree (AST), such parsers are designed to pinpoint, but not correct, syntax errors as they are encountered. While useful for compilation, such parsers are not useful when translating syntactically broken code.

DESCRIPTION

This disclosure describes a parser that is resilient to syntax errors in code, and which can, by applying local corrections, continue to parse the rest of the code after encountering a parse error.
Fig. 1: (A) Input code to the parser, which has syntax errors (B) Output code from the parser, which has corrected the syntax errors

Fig. 1 illustrates the parsing of broken code, per the techniques of this disclosure. Fig. 1(A) illustrates an example code snippet that has syntax errors (denoted by the red arrows). The syntax errors in this example are as follows: an if-block has not been ended; a for-loop has not
been ended; and the function definition has not been ended. The code in Fig. 1(A) is input to the parser. The parser produces as corrected output the code in Fig. 1(B), adding appropriate statements (denoted by green check marks) to correct the detected errors.

![A) Input code](image1)

![B) Output code](image2)

Fig. 2: (A) Input code to the parser, which has syntax errors (B) Output code from the parser, which has corrected the syntax errors

Fig. 2(A) illustrates another example of broken code which is fed as input to the parser, which generates corrected code as shown in Fig. 2(B). In this case, the function headers and their ends in the input code are not matched. Conventional parsers based on scoped parsing fail in such a case, and malware hidden in these functions can escape detection.

The parser as described herein parses one line of code at a time as an independent piece of code. These individual parsed lines are then combined along with the context and scope to create an abstract syntax tree. Errors in the code are localized to particular lines and corrected.
Once corrected, the code can be subject to malware detection prior to or after translation into the
target language.

If no correction is found for a particular erroneous line, the line of code is copied into the
target language as a comment statement. An additional clarificatory comment is added indicating
that that line was left uncorrected.

The described parsing techniques are resilient to syntax errors and can continue parsing
code by applying local corrections upon encountering a parse error. The techniques can be used
to facilitate standalone malware detection, e.g., malware detection without conversion to a target
language. For example, malware that is included as a spreadsheet that includes macros that have
intentionally broken code (which would bypass conventional malware detection) and is sent as
an email attachment. The broken code in the macros of the attachment is corrected, thereby
enabling malware detection.

The described techniques are a form of static analysis and offer advantages over other
techniques. Languages that are used for document/spreadsheet macros do not support arbitrary
execution over a string and malware is difficult to hide from static analysis. For example, other
techniques such as file hashing cannot deal with malware variants, regular expression analysis
often generates false positives and can miss certain type of code (e.g., hard to detect obfuscate
names/ usage).

While dynamic analysis using a sandbox (that can simulate the code) or a virtual machine
(that provides a real environment for code execution) can be performed to detect malware, such
techniques are expensive and have limitations (e.g., a sandbox requires a good script engine and
parser being available for the code, and simulation has limitations; it is difficult to deal with
cloaking, nondeterminism, cryptographic, and targeted malware using a virtual machine
implementation). Further, dynamic analysis requires significant computational resources and requires more time.

CONCLUSION
This disclosure describes a parser that is resilient to syntax errors in code, and which can, by applying local corrections, continue to parse the rest of the code after encountering a parse error. Once corrected, the code can be subject to malware detection prior to or after translation into the target language.

REFERENCES