Proceedings of the 2003 IEEE
Workshop on Information Assurance
United States Military Academy, West Point, NY June 2003

Insecure Programming: How Culpable is a Language's Syntax?

R. Chinchani, A. Iyer, B. Jayaraman and S. Upadhyaya
Dept. of Computer Science and Engineering
University at Buffalo, SUNY
Buffalo, NY 14260
Email: {rc27,aa44,bharat,shambhu}@cse.buffalo.edu

Abstract— **Vulnerabilities in software stem from poorly written code. Inadvertent errors may creep in due to programmers not being aware of the security implications of their code. Writing secure code is largely a software engineering issue requiring the education of programmers about safe coding practices. Various projects and efforts such as memory usage profiling, meta-compilation and typing proofs that verify correctness of the code at compile-time and run-time provide additional assistance in this regard. In this paper, we point out that in the context of security, one aspect that is perhaps underrated or overlooked is that vulnerabilities may be inherent in the syntax and grammar of a programming language itself. We leverage on some well-studied problems to show that small syntactic discrepancies may lead to vast semantic differences in programs and in turn, correlate to hard security errors. Our work will help caution programmers on the types of errors to avoid as well as serve as a guideline for language designers to lay emphasis not only on richness of language features but also the unambiguity of the syntax.**

Keywords— **Programming Language Security, Syntax Errors**

I. Introduction and Motivation

Security breaches of computer systems and data are often due to vulnerabilities in software. Buffer overflows [1] and heap corruption [2] are some common types of vulnerabilities. Programs that contain such vulnerabilities are at the mercy of an attacker's perseverance and creativity. Many of these vulnerabilities are due to undesirable side-effects of the features provided by popular programming languages. For example, pointers in the C programming language are very useful in accessing arbitrary memory locations. This is essential for producing systems-level code. On the other hand, the same flexibility provided by these pointers can be blamed for invalid memory accesses and resulting vulnerabilities.

Consequently, several research efforts have been invested towards securing programs while maintaining programming flexibility. Currently known techniques pertaining to programming language security can be generally categorized as: (1) proactive compile-time static analysis and detection, and (2) reactive assertion based runtime checking. Compile-time techniques [3] attempt to detect errors before actual execution of the program. These are often implemented as compiler extensions or separate tools. Runtime techniques [4], [5] use assertions or code annotations to detect violations during program execution.

Not all errors that have security implications can be *directly* attributed to pointers and invalid memory accesses. For example, consider the infamous FORTRAN bug [6] that found its way into some critical code. Since FORTRAN is not whitespace-sensitive, a loop construct was misinterpreted as a variable assignment resulting in erroneous computations. We give another example in C of such off-by-one errors resulting in valid but potentially dangerous statement constructions.
An inadvertent semi-colon on line #4 causes the 'for' loop to iterate over an empty statement. At the end of the loop, the variable 'i' has a value 100. Consequently when it is used as an array index to the variable 'buffer', it causes a buffer overrun.

```
1  int i;
2  char buffer[100];
3
4  for (i = 0; i < 100; i++) ;
5  {
6    buffer[i] = (char)NULL;
7  }
```

The correct code is as follows:

```
1  int i;
2  char buffer[100];
3
4  for (i = 0; i < 100; i++)
5  {
6    buffer[i] = (char)NULL;
7  }
```

These kinds of software bugs occur because there is not enough distance between two valid sentences in the language. Some errors are programming language-specific, and it may not be possible to replicate them in a different language. These errors are generally easy to commit and difficult to detect. The C programming language notoriously abounds in such errors; and languages that are based on C, such as C++ and Java, also inherit some of these undesirable properties.

In this paper, we develop a technique to evaluate the potential for such serious syntactic errors in popular languages such as C, Java and Perl. This technique is based on looking for certain

ISBN 0-7803-7808-3/03/$17.00

patterns in the syntax of a language such that a small syntactic change can cause a large change in the structure and control flow of the program. These changes may have some security implications as a result of buffer overruns and invalid memory accesses. Currently, we are developing a tool that automates this process. As security awareness grows, more and more emphasis is laid on good and safe programming practices. Our work finds practical use in this context. Programmers can be warned about the possibility of these errors in a particular language that they must produce code in.

A. Paper Organization

We discuss some relevant contemporary research in Section II to put our work in perspective. The main technique is discussed in Section III. We evaluate some popular languages in Section IV and report our results. We conclude the paper with discussion about the work accomplished and the future direction in Section V.

II. Background and Related Work

Some of the work that we present in this paper finds its basis in well-studied problems of software engineering and algorithm design.

A. Secure Software Engineering

Due to the nature of the data that they process, military establishments have traditionally required high assurance systems long before security became a mainstream concern. The U.S. effort of the "rainbow series" of books, including the "orange book," [7] aims to address these concerns. They specify security and corresponding verification requirements, placing systems into different security classes. Due to its heightened emphasis, recent research efforts have focused on bringing security into earlier cycles of software development. At the requirements engineering stage, high-level modeling tools such as UML [8] are being modified to relate security requirements with functional requirements. Aspect-oriented programming proves another interesting solution to implementing security policies throughout various pieces of a system while still isolating the concern [9]. As the role of software systems have shifted and expanded, the goal of sound software engineering practices are also shifting to create fault-tolerant, reusable, and *secure* code [10] [11].

This work finds a finds its place in both the design as well as implementation and testing phases of software engineering. At the design level, the specified security policy may influence the language chosen to implement a particular module. This tool will aid in making this choice. At the implementation and testing phases, this tool can caution programmers on the types of language-specific errors to avoid.

B. Existing Critiques of Languages

In general, languages are compared and contrasted based on their expressive power, feature set, and efficiency [12] [13]. C is known for its efficiency and flexibility, Java for its object-oriented nature and strong type system, Perl for its scripting power and regular expression manipulation. Beyond base constructions, the feature set of each of these languages is tailored to these specific qualities. As program security is becoming more of an imminent concern, the inherent security features of languages are being examined. Research efforts have been primarily focused on type systems, language flexibility, and the permissible feature set [14] [15] [16]. We seek to take a more low-level look at language security, and examine the implications of a language's abstract syntax and semantics.

C. Edit Distances

Given a language and the grammar based on which strings belonging to that language can be generated, the *minimum edit distance* between two strings is defined as the minimum number of deletions, insertions, or substitutions necessary to transform one string into the other. This notion was formalized by Wagner and Fisher [17] in 1974 and has largely been used in dictionary look-up algorithms [18], spell-checkers, and text auto-completion tools. Currently, the minimum edit distance of strings is being re-examined and has found applications in the field of syntactic pattern and speech recognition [19].

With respect to programming languages, edit distances are used in compiler design to automatically correct or recover from syntax errors. These compilers may repair an incorrect input string by performing a least-errors analysis and replacing it with valid strings of distance k away from that input string [20] [21]. The error-correcting compilers seek to transform invalid strings into valid sentences in a given language. In our paper, we seek to examine the edit distances between two *valid* sentences of a given language, a step beyond error-correcting compilers in the process of code analysis. We argue that very small edit distances between two valid sentences in a language may lead to hard-to-detect errors and possible security vulnerabilities.

D. Type Correctness

Type theory provides a formal framework to express and verify certain correctness properties of programs. Imposing types on all variables, functions, structures, etc., declared in a program, allows compilers to statically detect numerous common programmer errors. This level of ensuring program correctness has been employed to guarantee certain safety properties of programs. The system of enforced types can be used, for example, to provide a proof of program correctness at execution time [22] or secure information flow between variables at various degrees of privacy [23].

III. Evaluation Criteria

The primary focus of this work is to look at low level language features such as a language's syntax and semantics, and investigate whether certain security problems are rooted there. We identify and formalize certain criteria before proceeding to evaluate some popular programming languages.

ISBN 0-7803-7808-3/03/$17.00 © 2003 IEEE

Consider a programming language $\mathbb{L}$ with alphabet Σ. Let s be a string belonging to the language, then, $s \in \mathbb{L} \Rightarrow s \in \Sigma^*$, but the converse is not generally true. Every string in the language is generated using a set of rules called the syntax production rules and semantics of the language.

We define two tests to evaluate the potential for programmer errors.

1. *Simple Hamming Distance Test*

Let $\mathcal{K}$ be a set of terminal symbols such that $\mathcal{K} \subseteq \Sigma^*$. Let x and y be two terminal symbols belonging to $\mathcal{K}$. Then, $dist(x,y)$ represents the hamming distance in terms of symbols rather than bits. For example, $dist$('+', '++') = 1 and $dist$('if', 'for') = 3. The set of terminals $\mathcal{K}$, represents the keywords and basic building blocks of a programming language. In order to prevent one valid keyword or symbol from being easily transformed into another valid keyword or symbol, at least the following should hold true.

$$\forall t_i, t_j \in \mathcal{K} \text{ and } t_i \neq t_j,\ dist(t_i,t_j) > 1 \qquad (1)$$

It can be easily seen that this is a weaker kind of test. Although the distance between symbols or keywords may be one, replacing one by the other does not always result in another valid phrase. For example, consider the statements 'i++' and 'i+='. The terminal '++' is replaced by '+=' and while the first phrase is valid, the second one in isolation is not. Therefore, property (1) is not sufficient since invalid phrases may be flagged as off-by-one syntactic changes. This test can raise too many such false positives.

2. *Substring Construction and Comparison Test*

The syntax rules of a programming language form a directed AND-OR graph with a single root node, which is the start symbol. The edges represent valid productions and non-terminal expansions. The leaf nodes typically contain the terminal symbols. Recursion in the grammar rules generates cycles in the graph. Valid strings or programs are obtained by expanding these rules starting from the root node or the start symbol. Consider the following grammar,

$$\begin{aligned} S &:= A \mid B \\ A &:= XY \\ B &:= ab \\ X &:= a \\ Y &:= bc \end{aligned}$$

The graphical representation of this grammar is given by Fig. 1.

Expanding using production rules only ensures syntactic correctness of phrases, but not all generated phrases are meaningful. At one step higher, the semantic rules are more context-sensitive and serve to limit the set of valid productions and non-terminal expansions.

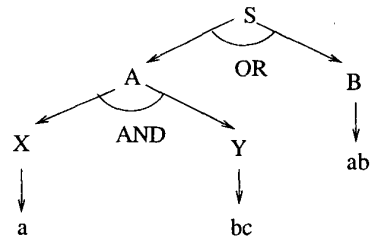

Fig. 1. A graphical representation of a simple grammar

Clearly, generation of all possible strings or equivalent programs in a general-purpose language is not feasible or practical because of the exponential explosion in their number. We employ some heuristics to keep the number of strings manageable. After constructing the directed graph, we expand the non-terminals using bottom-up traversal, starting from the leaf nodes. This results in the generation of substrings or phrases in an incremental fashion. We may encounter two kinds of rules when generating these substrings: (1) rules that immediately expand to terminals, and (2) rules that are recursive. We begin expanding rules of the first kind. After d such iterations, we have generated all possible terminal substrings at a depth d starting from the leaf nodes. We handle recursion by limiting the substring generation process tc produce only value-added strings. Consider the following production rules:

$$\begin{aligned} S &:= Sa \\ S &:= b \end{aligned}$$

Given a and b are terminals, the only value-added strings to be generated is the set $\{a, ab, abb\}$. Recursing these rules any further only results in redundancy for the purposes of our off-by-one string comparison. For example, the structure of the C statement 'x = a + b + b + b + b ...;' is effectively captured by 'x = a + b;'. Since not all phrases that are generated are meaningful, we prune them using semantic rules.

Disjunction in the digraph at some point means that a single non-terminal can be expanded in many ways and one expansion can be replaced by the other. All that remains to be determined is whether these substrings are close enough to cause an inadvertent programming error. Of particular interest are those replacements that result in large structural changes in a program.

IV. Security Implications

To evaluate the efficacy of our tool, we have compared three commonly-used general purpose languages: C, Java, and Perl. To each of these languages, we apply the two tests described above: the Simple Hamming Distance Test and the Substring Construction and Comparison Test. An overview of the evaluation process is given in Fig. 2.

Certain patterns of errors emerge that provide an insight into

ISBN 0-7803-7808-3/03/$17.00 © 2003 IEEE

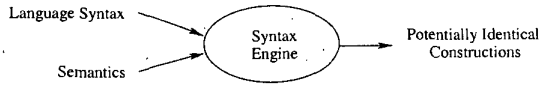

Fig. 2. Syntax-directed processing engine

the relationship between small syntactic changes and their security implications.

A. Similar Variable Names

When variable names that represent buffers and pointers to memory locations are very similar, a small syntactical change may transform one variable name to another, resulting in an invalid memory access. The following piece of code illustrates this effect.

```
1   char *buf1;
2   char *buf2;
3
4   buf1 = malloc(...);
5   strcpy(buf2, "bad string copy");
```

These errors are more or less language-independent and cannot be attributed to an inherent flaw in the syntax of any specific programming language. Although our tool reports such errors, we recognize that they are primarily due to carelessness on the part of a programmer.

B. Operator Switching

The syntactical notation of operators in C can cause small changes very easily. For example, 'i += j' may be changed to 'i -= j'. If the variable 'i' is used as an index into an array, it is quite possible that the array bounds are violated. Since Java and Perl have a similar syntax notation to that of C, this pattern of errors is common to all of these languages. Another example of interest involves the operators '==' and '='. The former is a boolean conditional operator while the latter is an assignment operator. This is the source of many errors in the conditional part of the 'if' statement. Java has eliminated this by requiring only a boolean operator in an 'if' statement. Perl is often the programming language of choice for CGI scripts. It is renowned for its mastery of text processing through strong support for regular expressions. However, numerous syntax-directed errors may be the result of this regular expression manipulation. A difference of one symbol within a regular expression can yield vastly different results than expected.

```
1   if (m/$username/[fF]red/) { ... allow access ... }
2
3   if (m/$username/[f-F]red/) { ... allow access ... }
```

The condition in line #1 allows access only when username matches either 'Fred' or 'fred', whereas the condition in line #3 inadvertently allows access to other users.

C. Overloaded Terminal Symbols

Consider the operators '&' and '*' in C. '&' is used as a bitwise binary operator as well as the 'address-of' unary operator. Similarly, '*' is used both for multiplication and dereferencing. Since C is a weakly typed language, inadvertent typecasts may occur. The following piece of code illustrates this error.

```
1   int i, j, k;
2   int rightsize, wrongsize
3   int *p;
4
5   rightsize = i * j & k;
6   p = malloc(sizeof(int) * rightsize);
7
8   wrongsize = i * & k;
9   p = malloc(sizeof(int) * wrongsize);
```

The right hand sides of statements in lines #5 and #8 are different only in one symbol, but they compute two very different values. The variable 'i' is multiplied with the address of the variable 'k' and the variable 'wrongsize' in line #8 is very likely to be assigned a smaller value due to an integer overflow. Any use of this memory thus incorrectly allocated could result in a program crash. Such errors are less likely to happen in Java due to a stronger notion of type correctness and absence of pointers.

D. Ambiguous Syntactic Constructs

The semi-colon in C acts as a statement terminator. It has the capability of ending a statement and marking the beginning of another. It is very likely that insertion of semi-colon in a piece of C code can cause drastic changes in the program's structure and control flow. An example is presented in Section I. The syntactical construction of the 'for' statement is as follows:

$$\text{for}\,(\,expression_{opt}\,;expression_{opt}\,;expression_{opt}\,)$$
$$statement$$

A snapshot of the graphical representation of the grammar involving the non-terminal symbol *statement* is given by Fig. 3. Here, the non-terminal 'statement' may be expanded in multi-

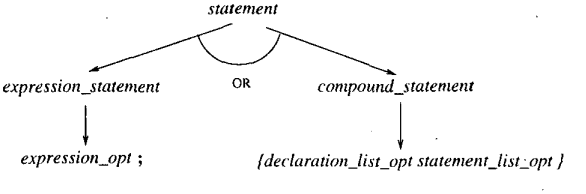

Fig. 3. Expansion of the non-terminal symbol *statement* in C syntax

ple ways. If the left subtree is used, then 'statement' expands to just ';'. On the other hand, the right subtree expands to '{ body }'. As statements can appear sequentially, both of the following

ISBN 0-7803-7808-3/03/$17.00 © 2003 IEEE

expansions are valid.

for(...) *compound_statement*
for(...) *expression_statement compound_statement*

These subtle differences in syntax expansions lead to a string difference of one between the two blocks of code in Section I. This induces vastly different program behavior with potentially serious security implications. Java suffers from this type of error as well. Since this error is not type-specific, it eludes even a strongly-typed system. Perl prevents this type of error by requiring that a 'for' loop declaration be followed by '{ body }'.

We have observed that the semi-colon also causes some strange constructs in the C programming language. The following piece of code illustrates one such scenario:

```
1  int i, j, k, l;
2  int size;
3  int *p;
4
5  size = i > j ? k : l;
6  p = malloc(sizeof(int) * size);
7
8  size = i > j ; k : l;
9  p = malloc(sizeof(int) * size);
```

The ternary operator is used in line #5, but in line #8, a difference of only one symbol alters the semantics significantly. The variable 'k' is now a label and the variable 'size' is assigned a boolean value. As a result, the amount of memory is not allocated as intended.

V. Conclusion and Future Work

In this paper, we have enumerated some criteria for evaluating the culpability of a language's syntax and grammar in the context of insecure programming. The syntax and semantics represent the expressive power of a programming language. While designing a language, focus is predominantly laid on the features of a language. However, the task of determining a proper syntactic notation, while mundane, is also very important. With a limited set of symbols, chances of syntactic collisions and errors are greatly increased. Our suggestion to language designers is to provide a less ambiguous syntax, as this will prevent inadvertent errors. Also, based on our findings, we make the following suggestions to programmers:

1. Variable names that are very similar to each other or the keywords should be avoided.
2. Care must be taken when using operators, especially in C, because an unintended operator switch may result in an undesired typecast.
3. White space should be well-utilized in code, thereby reducing the number of unexpected interactions between operators.
4. Also, more caution should be exercised when working in the context of weakly-typed languages, such as C. The lack of a type system leaves more syntactic ambiguity with respect to valid constructions of the language.

The number of production rules and recursive grammar used to describe the syntax of these general-purpose languages currently limit our evaluation capability. Also, depending on the language, a single sentence can have many alternate syntactically valid constructions, which are off by one symbol. But not all of them are meaningful phrases and this raises too many false positives.

Some parts of the evaluation process currently require manual intervention. Our short-term goal is to identify immediate issues and completely automate the process.

On the long run, we speculate that this tool can be integrated into modern rapid development tools or program development environments. As programs are written, they could be dynamically checked for potential erroneous constructions. Similar to error-correcting compilers, a stochastic model could be applied to transform (or suggest a transformation thereof) a possibly erroneous construction into a more likely sentence that is a small distance away. Such a tool will greatly reduce the number of errors which fall through the previous levels of analysis.

VI. Acknowledgments

This research was supported in part by National Security Agency through Grant No. MDA 904-02-1-0215.

References

[1] A. One, "Smashing the Stack for Fun and Profit." Phrack 49, Vol. 7, Issue 49, November 1996.
[2] "Wu-Ftpd File Globbing Heap Corruption Vulnerability," 2001. http://www.securiteam.com/unixfocus/6U00V0035Q.html.
[3] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken, "A First Step towards Automated Detection of Buffer Overrun Vulnerabilities," in *Network and Distributed System Security Symposium*, (San Diego, CA), pp. 3–17, February 2000.
[4] C. Cowan, C. Pu, D. Maier, H. Hinton, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang, "StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks," in *7th USENIX Security Symposium*, (San Antonio, TX), January 1998.
[5] C. Cowan, M. Barringer, S. Beattie, G. Kroah-Hartman, M. Frantzen, and J. Lokier, "FormatGuard: Automatic Protection From printf Format String Vulnerabilities," in *10th USENIX Security Symposium*, (Washington DC), August 2001.
[6] M. Brader, "Fortran Story - The Real Scoop." Forum on Risks to the Public in Computer and Related Systems, Vol. 9 #54, ACM Committee on Computers and Public Policy, 1989.
[7] "Trusted computer security evaluation criteria," *DOD 5200.28-STD*, 1985.
[8] J. Jfurjens, "Towards development of secure systems using UML," in *Fundamental Approaches to Software Engineering (FASEiETAPS, International Conference)*-(H. HutJmann, ed.), LNCS, Springer, 2001.
[9] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-oriented programming," in *Proceedings European Conference on Object-Oriented Programming*, vol. 1241, pp. 220–242, Berlin, Heidelberg, and New York: Springer-Verlag, 1997.
[10] P. T. Devanbu and S. G. Stubblebine, "Software engineering for security: a roadmap," in *ICSE - Future of SE Track*, pp. 227–239, 2000.
[11] I. Jacobson, M. Griss, and P. Jonsson, "Software reuse: Architecture, process and organization for business reuse," 1997.

[12] M. Felleisen, "On the expressive power of programming languages," in *ESOP '90 3rd European Symposium on Programming, Copenhagen, Denmark* (N. Jones, ed.), vol. 432, pp. 134–151, New York, N.Y.: Springer-Verlag, 1990.

[13] L. Prechelt, "An empirical comparison of seven programming languages," *Computer*, vol. 33, no. 10, pp. 23–29, 2000.

[14] D. Kozen, "Language-based security," in *Mathematical Foundations of Computer Science*, pp. 284–298, 1999.

[15] D. M. Volpano and G. Smith, "A type-based approach to program security," in *TAPSOFT*, pp. 607–621, 1997.

[16] F. B. Schneider, G. Morrisett, and R. Harper, "A language-based approach to security," *Lecture Notes in Computer Science*, vol. 2000, pp. 86–??, 2001.

[17] R. Wagner and M. Fischer, "The string-to-string correction problem," *Journal of the ACM (JACM)*, vol. 21, no. 1, pp. 168–173, 1974.

[18] R. Baeza-Yates and G. Navarro, "Fast approximate string matching in a dictionary," in *String Processing and Information Retrieval*, pp. 14–22, 1998.

[19] E. S. Ristad and P. N. Yianilos, "Learning string-edit distance," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 20, no. 5, pp. 522–532, 1998.

[20] G. Lyon, "Syntax-directed least-errors analysis for context-free languages: a practical approach," *Communications of the ACM*, vol. 17, no. 1, pp. 3–14, 1974.

[21] S. Graham and S. Rhodes, "Practical syntactic error recovery," *Communications of the ACM*, vol. 18, pp. 639–650, November 1975.

[22] G. C. Necula, "Proof-Carrying Code," in *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Langauges (POPL '97)*, (Paris), pp. 106–119, Jan. 1997.

[23] G. Smith, "A new type system for secure information flow," in *CSFW14*, IEEE Computer Society Press, Jun 2001.

ISBN 0-7803-7808-3/03/$17.00