Separation of Concerns for Security

John Viega
Reliable Software Technologies
viega@rstcorp.com

David Evans
University of Virginia
Department of Computer Science
evans@cs.virginia.edu

Abstract

Writing secure code is something most developers know little about. As a result, software vulnerabilities are quite common. We postulate that, by isolating security as a separate concern, this problem can be alleviated.

1 Introduction

We propose to solve some of the problems with developing secure software by separating out security as a concern.[1] Doing so will allow security policies to be separate from the code, allowing developers to write the main application, and a security expert to specify security properties. Also, we believe that a well designed policy language can abstract away much of the expert knowlege currently required for writing secure, allowing developers to reasonably secure their code, even if a security expert is not available to assist.

Our approach is general enough to be applied to any language, and in the future we anticipate experimenting with versions for several different languages. At present, we focus on C because its security vulnerabilities are well known and it is a relatively small and simple language. Further, despite its flaws, even today C is still widely used to write security-critical applications.

2 Motivation

Experience has shown that developers aren't very good at writing secure software. Part of the problem is a lack of education; few classes cover this material, and no books cover it well. However, education isn't the entire issue, as evidenced by the fact that buffer overflow exploits in C code are quite common, even in software written by developers who know a lot about this problem, including which functions in common libraries are potentially dangerous.

We think it should be easier to design security into an application, instead of relying on the "penetrate and patch" approach to security, where problems are addressed in an ad hoc manner, generally as flaws are revealed in the field. Unfortunately, the "penetrate and patch" approach to security is widespread, as opposed to the alternative of designing software with security in mind. There are two primary reasons why this problem is so prevalent. The first is that no coherent design time methodologies or tools are widely available. In fact, there are no comprehensive resources available to help write secure programs; developers do not know what the problems are. As a result, developers continue to write insecure code and to fix security problems when someone happens to notice them. The second problem is that designing and implementing secure systems currently requires a lot of expert knowledge. Even with good methodologies, good tools are still critical to alleviating this problem, as the average developer is likely to be either unwilling or unable to use the methodology effectively if no tools are available.

Another large part of the problem is the state of programming languages from a security point of view. A few languages have given significant thought to security primitives that should be present to help programmers write better code. However, two of the most widely used programming languages present significant security risks (specifically, C and C++). This is because many of their standard features can easily be used in such a way to inadvertently leave a security flaw in a program. Even languages with significant security architectures such as Java leave something to

[1]Our work is being funded under the DARPA IASET program.

be desired; we have found in practice while looking at many commercial products that a large percentage of applications have significant security problems that are present in the design phase, and will persist through to implementation despite the language used. Some of the more common problems include misuse of security protocols and an unrealistic view of what a system should consider trusted. Languages have yet to make significant inroads in this area.

3 Goals

In talking with many developers about security, we have noticed that even security-aware people find it much easier to write their program, and then later go back and try to "bolt-on" security as an afterthought. In our experience, this approach doesn't work; security currently generally needs to be designed into an application from the beginning. There is a fundamental conflict here between what works well in practice, and the way developers work, even when they are well educated on security matters.

All tools we have encountered to help prevent security vulnerabilities in general-purpose programming languages are after the fact tools, such as vulnerability analysis tools that look for common programming and configuration errors. They do not address how a developer should design and implement software in a security conscious manner. We believe that a more proactive approach is appropriate. A good tool for solving these problems would meet the following goals:

1. The amount of expert knowlege necessary to secure source code should be minimized. It should not be easy for a developer to accidentally introduce security problems into a program just because the language and the concepts of secure programming haven't been mastered. Common language pitfalls should be averted, and the programmer should be protected from common paradigms of failure that are not language specific.

2. The security-related elements in a program should be abstracted out of the program proper, for the sake of clarity, maintainability and reuse.

3. Security policies should be defined using a general enough language that it is possible to create new policies to deal with application-specific issues or previously unknown security vulnerabilities.

4. An emphasis should be placed on "security by default", so that the level of effort for developing secure C and C++ applications is minimized. This is a bigger challenge for C than for most other languages, since the language and standard libraries contain so many unsafe operations. Buffer overflows are one well known problem. Others include file access calls that are susceptible to security-critical race conditions, as well as most uses of the `popen` and `system` library calls.

5. Currently, if a security consideration is omitted in just *one* place, it can easily lead to a flaw. It should be easy to express policies about the program that apply generically to a consideration, and then have the policy be applied through the program automatically. For example, it should be possible to say that no secret information is ever sent over the network unencrypted (perhaps the actual implementation may choose to encrypt all data to satisfy this property).

6. Legacy source code with known or potential security problems should be able to benefit from such a tool; the amount of new code necessary should be minimized.

7. When it makes sense to do so, security policies should be reusable across different applications.

4 An Aspect-Oriented security approach

Seperation of concerns can help meet all of the above goals. A specification language can be defined that will be woven with a C application, separating security code from the rest of the program. This language could have reasonable defaults that can be overridden, and could have some degree of polymorphism (e.g., the programmer should be able to define classes of variables, place variables in those classes on a case by case basis and specify what kind of buffer overflow protection each class should be afforded). Such polymorphism would also help afford reuse of specifications across applications; the security aspect language could allow two types of specifications: those that map variables to symbolic names (program specific), and those that define security policies based on symbolic names.

We are in the process of designing and building a specification language intended to meet the above goals. Specifications written in this language will be woven together with C programs to produce secure programs. Our approach will be to take the security specification and the C source, and "weave" the two into a single C program.

some of the problems we are addressing include:

1. Buffer overflows.

2. Environmental attacks (Trojan environment variables).

3. File-based race conditions.

4. Randomness attacks (guessing security-critical "random" numbers).

5. System call failures.

As an example of our approach, we will discuss system call failures. Some attacks exploit system calls failing in ways not anticipated by the programmer. When those calls fail in an unexpected way, default handlers should be called that perform reasonable actions. We will assume that the return values for individual calls are not checked in the program proper, unless a static analysis can determine otherwise. In the specification language, programmers should be able to specify a policy for all such calls, for single calls, and for classes of calls. For example, one should be able to say, "for every memory allocation call that fails, abort". It should be possible to override this behavior at individual sites, as well. The actual weaving is a simple transformation. For example, consider the following code that allocates 256 characters:

```
char *p;
p = (char *)malloc(256*sizeof(char));
```

A typical security specification would direct the weaver to convert this code to the following:

```
char *p;
p = (char *)malloc(256*sizeof(char));
if(p == NULL) {
  log("Out of memory.");
  abort();
}
```

We estimate that these kinds of problems cover at least 90% of all security incidents in C programs over the past 3 years (Buffer overflows alone account for 50% of the problem [10]).

We would also like to build features that ease the burden of using other security primitives, such as:

1. Automatic logging.

2. Specifying critical sections.

3. Authentication and Encryption.

For example, network connections can be authenticated and encrypted automatically via code instrumentation. In the case of authentication, we anticipate two options. In one option, the program will need to authenticate itself with a remote server. The programmer would have to specify the type of authentication expected by the remote server, and a means by which to read the authentication data (e.g., a passphrase). The second option is that the program is expecting remote authentication. In this case, the programmer specifies the type of authentication, and also specifies a database (such as a keystore) against which the remote connection should be authenticated. The actual implementation will just add approppriate code surrounding those calls that establish a network connection. The implementation of encrypted network connections will be quite similar, except that reads and writes to a network connection will also need to be modified to actually encrypt and decrypt.

5 Related work

The space of available software security assurance is currently inhabited primarily by small, open source tools that only address a fraction of the actual problem. There are multiple patches for the gcc compiler that implement array bounds checking. There are several tools that provide some sort of security against buffer overflow attacks, including StackGuard[2] and FIST[5]. However, most of these tools are solely interested in buffer overflows.

Another type of tool in the security assurance domain is the "secure data-flow" tool. Examples of this tool are the "taint" version of Perl, and the JFlow programming language (a Java extension) [7]. In such tools data are labeled either "untrusted" or "trusted". "Untrusted" data cannot be passed to trusted items without the programmer explicitly allowing it. Similarly, "trusted" data cannot be passed to "untrusted" items for fear of leaking secret information, unless explicitly declassified by the programmer.

All of the capabilities of the above tools could be described in a security specification and woven into a program using our aspect-oriented security approach. We hope to incorporate existing tools as off the shelf technology whenever possible. For example, the aforementioned tools for preventing buffer overflows can potentially be leveraged in the implementation of our approach. We hope to provide a uniform and general purpose interface to these tools, while adding a large amount of flexibility and extensibility; only a few of the many problems we seek to handle are addressed by current tools.

Commercial tools in the security assurance space are almost universally general-purpose, and not security-specific. For example, there are many tools such as Rational's Purify that can help find and fix buffer overflow problems, even though the tool is not specifically a security tool[6].

Another class of tools are after-the-fact tools that support the "penetrate and patch" model. These tools generally are concerned with taking preexisting source code, and identifying potentially dangerous constructs based on a database and some static analysis. Currently, the only publicly available tool for source code analysis is `ITS4`, which scans C and C++ code for over 100 potential problems[9]. Wagner has a buffer-overflow scanner that performs a more sophisticated analysis; however it is not publically available, and is limited in scope[10]. Similar tools exist that are general purpose, and may catch some security bugs, including lint tools such as LCLint [3].

Previous work has also been done in policy languages for security. Most such languages specify file access control, allowing the programmer to give explicit policies stating what a program can and cannot do to files. Examples of such systems include Naccio[4], Ariel[8] and PolicyMaker[1]. We anticipate incorporating this sort of tool as a small part of our total functionality.

6 Conclusion

We have identified some of the major problems plaguing software security, and discussed how separating security from a program proper might help alleviate these problems. We are currently working on designing a language for defining security concerns and implementing a weaver that generates and integrates code into C programs. There are many currently open questions we plan to address in our work, including:

1. What properties are required of a language for describing security policies that can be woven into a program?
2. What class of security policies can be enforce using a separation of concerns approach?
3. What opportunities will there be to analyze security specifications to make claims about programs that have been woven into a single program?

References

[1] M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized trust management. In *Proceedings of the 17th IEEE Symposium on Security and Privacy*, May 1996.

[2] C. Cowan et. al. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the Seventh USENIX Security Symposium*, pages 63–77, San Antonio, TX, 1998.

[3] D. Evans, J. Guttag, J. Horning, and Y. Meng Tan. LCLint: A tool for using specifications to check code. In *Proceedings of the SIGSOFT Symposium on the Foundations of Software Engineering*, December 1994.

[4] D. Evans and A. Twyman. Flexible policy-directed code safety. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, May 1999.

[5] A. Ghosh, T. O'Connor, and G. McGraw. An automated approach for identifying potential vulnerabilities in software. In *Proceedings of the 1998 IEEE Symposium on Security and Privacy*, May 1998.

[6] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the Winter USENIX Conference*, 1992.

[7] A. Myers. Practical mostly-static information flow control. In *Proceedings of ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, San Antonio, TX, January 1999.

[8] R. Pandey and B. Hashii. Providing fine-grained access control for mobile programs through binary editing. Technical Report CSE-98-8, University of California, Davis, 1998.

[9] J. Viega, J.T. Bloch, T. Kohno, and G. McGraw. Its4: A static vulnerability scanner for C and C++ code. In *Submitted to USENIX Security*, 2000.

[10] D. Wagner, J. Foster, E. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Proceedings of the Year 2000 Network and Distributed System Security Symposium (NDSS)*, pages 3–17, San Diego, CA, 2000.