THE IMPACT OF PROGRAMMING LANGUAGES ON THE SOFTWARE'S SECURITY

Graduate Student: Alexandru Gavril Bardas
James Madison University
Computer Science - Secure Software Systems

Intoduction

Security is usually defined as the ability of a system to protect itself against accidental or deliberate intrusion[1]. Ensuring integrity, confidentiality, availability, and accountability requirements even in the presence of a determined, malicious opponent is essential for computer security. Sensitive data has to be manipulated and consulted by authorized users only (integrity, confidentiality). Furthermore, the system should resist "denial of service" attacks that attempt to render it unusable (availability). Also the system has to ensure the inability to deny the ownership of prior actions (accountability).

In more colorful language, computer security has been described as "programming Satan's computer[3]": the designer must assume that every weakness that can be exploited will be[2].

Security is a characteristic of a complete system, and involves many different topics, both computer-related (hardware, systems, networks, programming, cryptography) and user-related (organizational, social policies and laws)[2]. While system designers typically develop secure systems architectures at higher levels than the programming languages; the principal contribution of programming languages to the security of the whole system is low- and intermediate-level support for high-level abstractions[4].

This paper presents some of the proprieties of programming languages that affect the security of the whole system.

Security in Today's world

Security became very important in many industries today, and every organization must adopt proactive security measures for data, processes, and resources throughout the information life cycle. This is the reason why I think that an organization must have a thorough understanding of the business challenges related to security, critical security threats, exploits, and how to mitigate risk and implement safeguards and countermeasures. Adopting security by using proactive approaches and implementing correctness by construction becomes essential to organizational health and well-being. As a result operational efficiency and cost effectiveness may be increased significantly.

Modern programming languages aim to support a faithful and robust implementation of high-level security ideas and give system designers confidence in their underlying programming models safety and reliability, and a flexible and expressive way to precisely specify and rigorously enforce high-level security policies[4].

Programming Languages Security Features and Exploits

C/C++

Perhaps the most frequently used low level programming languages are C and C++. Type checking contributes to the overall safety of the languages but C and C++ are not strongly typed and the parameter type checking can be avoided by using unchecked aliasing in unions. Also the memory safety in C/C++ is not ensured in all the functions of the language. The C exploits such as buffer overflow and format-string attacks are well known security issues.

A buffer overflow occurs in situations when data is written outside of the boundaries of the memory allocated to a particular data structure[5]. There are several reasons that result in buffer overflows in C and C++:

a. Defining strings as a null-terminated arrays of characters[5]
b. Implicit bounds checking is not performed in C and C++[5]
c. Providing standard library calls for strings that do not enforce bounds checking

These properties have proven to be a highly reactive mixture when combined with programmer ignorance about vulnerabilities caused by buffer overflows.

Buffer overflows can go undetected during the development and testing of software applications. Common C and C++ compilers do not identify possible buffer overflow conditions at compilation time or report buffer overflow exceptions at runtime[5]. Therefore dynamic analysis tools are not too useful unless the tested data enforces a buffer overflow.

However not all buffer overflows lead to an exploitable application vulnerability. A buffer overflow can cause a program to be vulnerable to attack when the program's input data is manipulated by a user. Even buffer overflows that are not obvious vulnerabilities can introduce risk.

All in all buffer overflows are a primary source of software vulnerabilities. Type unsafe languages, such as C and C++, are especially prone to such vulnerabilities. Exploits can and have been written for Windows, Linux, Solaris, and other common operating systems and for most common hardware architectures, including Intel, Sparc, and Motorola[5].

Format-string attack is a kind of injection attack, generally specific to C and C++[6]. Format string bugs come from the same dark corner as many other security holes: The laziness of programmers[7]. For instance writing *printf(s)* instead of *printf("%s", s)* can be described as a huge opened back-door for attackers. Format strings support '%' – delimited markup sequences for printing formatted data. In general these are output related markups that read data from memory for output. There is also *%n* which supports writing data to a reference-passed variable. Attackers could insert markup to cause the application to interpret their markup as legitimate application directives. The consequences can be really severe if such vulnerability is discovered. Application host compromise via attacker controlled arbitrary code execution. DoS (Denial of service) from application use because of memory corruption/crash and loss/corruption of data are some of the possible consequences.
While the attacks may seem subtle or unlikely, they are not difficult in practice. Once the vulnerability is discovered it is reliable to exploit[6].

Buffer overflow and format-string attacks can be avoided by using secure design and programming patterns. One common mitigation strategy is the adoption of new libraries that provide alternatives (a more secure approach to string manipulation). For instance the safe library functions from ISO/IEC TR 24731 are more secure and designed as easy drop-in replacement functions for existing calls. "ISO/IEC TR 24731 provides alternative functions for the C Library (as defined in ISO/IEC 9899:1999) that promote safer, more secure programming"[8].
In conclusion, it is recommended to use these functions in order to reduce the likelihood of vulnerabilities in the code.
To eliminate vulnerabilities that allow a format-string attack to be successful, one of the mitigation strategies is the use of statically defined strings for any variable and arguments calls. In case the strings are not statically defined the programmer has to ensure all constituent data sources are provided and fixed at build-time[6].

However there will be always a trade-off between convenience and security in the process of designing and implementing secure code. Most of the more-secure functions have more error conditions while less-secure functions are focused more on functionality. The choice of the libraries is also constrained by programming language choice, platform, and portability issues.

There are practical mitigation strategies that can be used to help eliminate vulnerabilities resulting from buffer overflows. It is not practical to use all of the avoidance strategies because each has a cost in effort, schedule, or licensing fees. However, some strategies complement each other nicely. Static analysis can be used to identify potential problems to be evaluated during source code audits. Source code audits share common analysis with testing, so it is possible to split some costs. Dynamic analysis can be used in conjunction with testing to identify overflow conditions. Runtime solutions such as bounds checkers, canaries, and safe libraries also have a runtime performance cost and can also conflict. For example, it may not make sense to use a canary in conjunction with safe libraries because each performs more or less the same function in a different way[5]. Buffer overflows are the most frequent source of software vulnerabilities and should not be taken lightly.

Java

The designers of Java were very careful so that the language does not offer vulnerabilities that lead to buffer overflows and other well known security vulnerabilities. The “Java language itself is type-safe and provides automatic garbage collection, enhancing the robustness of application code. A secure class loading and verification mechanism ensures that only legitimate Java code is executed”[10].

Java provides automatic memory management, garbage collection, and range-checking on arrays. These facts reduce the overall programming burden placed on developers, leading to fewer subtle programming errors and to safer, more robust code. Also the Java language defines different access modifiers (public, private, protected) that can be assigned to Java classes, methods, and fields, enabling developers to restrict access to their class implementations.

Java contains a set of APIs spanning major security areas, including cryptography, public key infrastructure, authentication, secure communication, and access control. In this way developers can easily integrate security in their code. Sun’s official website states that these APIs were designed around the following principles:

- Implementation independence
- Implementation interoperability
- Algorithm extensibility

Java also includes security providers. A good example in this sense is the *`java.security.Provider`* class which encapsulates the notion of a security provider in the Java platform. It specifies the provider's name and lists the security services it implements[10]. Also certain aspects of Java security, including the configuration of providers, may be customized by setting security properties.

The cryptography architecture in Java is a framework for accessing and developing cryptographic functionality for the Java platform. It contains APIs for a large variety of cryptographic services, including message digest algorithms, digital signature algorithms, symmetric bulk encryption, symmetric stream encryption, asymmetric encryption, password-based encryption (PBE), Elliptic Curve Cryptography (ECC), key agreement algorithms, key generators, Message Authentication Codes (MACs), (Pseudo-)random number generators. The Java platform also provides APIs and an implementation of the SSL and TLS protocols (for data that travels across the network) that provide functionality for data encryption, message integrity, server authentication, and optional client authentication[10].

"Authentication is the process of determining the identity of a user"[10]. Java provides APIs that allows an application to perform user authentication. Applications call into the `LoginContext` class (in the `javax.security.auth.login` package), which in turn references a configuration. The configuration specifies which login module (an implementation of the `javax.security.auth.spi.LoginModule` interface) is to be used to perform the requested authentication.

The access control feature in the Java platform protects the access to sensitive resources (for example, local files) or sensitive application code (for example, methods in a class). All the access control decisions are mediated by a security manager, represented by the `java.lang.SecurityManager` class[10].

The stack-inspection[4], type safety, automatic memory management and range-checking on arrays are some of the features that protect the applications against buffer overflow and stack smashing vulnerabilities. Despite of this complex mechanism, in January 2007 developers discovered a vulnerability that leads to a buffer overflow in Java[9]. The vulnerability has been reported in Sun Java Runtime Environment (JRE), which can be exploited by malicious people to compromise a vulnerable system. The vulnerability was caused due to an error when processing GIF images and can be exploited to cause a heap-based buffer overflow via a specially crafted GIF image with an image width of 0. Successful exploitation allowed execution of arbitrary code. The vulnerability was reported in the following versions:

- JDK and JRE 5.0 Update 9 and prior
- SDK and JRE 1.4.2_12 and prior
- SDK and JRE 1.3.1_18 and prior

Sun fixed the leak and the program update to a fixed version resolved the issue. Format-string attacks are not possible in Java because of the design of the language.
Also sandboxing can be achieved in Java (and not only in Java, CLR language platforms too) by combining different "lower-level language safety and security, for instance type safety and stack-inspection access control"[4].

As described above, one of Java's main design considerations is to provide a secure environment for executing mobile code. While the Java security architecture can protect users and systems from hostile programs downloaded over a network, it cannot defend against implementation bugs that occur in *trusted* programs[12]. Such bugs can open back-doors that facilitate the leak of private information, the abuse of privileges, and ultimately the access of sensitive resources by unauthorized users. To minimize the likelihood of security vulnerabilities caused by programmer error, Java developers should program respecting Sun's and CWE's recommended coding guidelines.

C#

Another example of programming language that has a lot of security features embedded in its architecture is C#. These features include permissions, type-safety, security policy, principals, authentication, and authorization.
The .NET Framework provides code access security and role-based security to help address security concerns about mobile code and to provide support that enables components to make decisions about what users are authorized to do. These security mechanisms are designed to have a simple, consistent model so that developers familiar with code access security are able to use role-based security, and vice-versa[13].

There are three kinds of permissions in C#: Code permissions, Identity Permissions & Role- based permissions. The namespace *System.Security.Permissions* is responsible for it and provides support in implementing custom permission classes also.

Type-safe code is able to access only the memory locations it is authorized to access. In case the code that is not verifiably type-safe, it can attempt to execute if security policy allows the code to bypass verification.

The security policy is the configurable set of rules that the Common Language Runtime (CLR) follows when it decides what it will allow code to do. Administrators set security policy, and the CLR enforces it[13].

A principal represents in C# a user or an agent that acts on the user's behalf. *.NET Framework* role-based security has support for three kinds of principals: Generic principals, Windows principals and Custom principal.

Authentication is the process of discovering and verifying the identity of a principal by examining the user's credentials and validating those credentials against some authority[13]. On the other hand authorization is the process of determining whether a principal is allowed to perform a requested action.

C# provides also besides the security features, security tools. These are command-line utilities that help perform security-related tasks and test components and applications before they will be deployed.

For instance Assembly Generation Utility (al.exe) takes as its input one or more files that are either in MSIL format or are resource files and outputs a file with an assembly manifest[13]. Other examples of security features are Code Access Security Policy Utility (caspol.exe), Software Publisher Certificate Test Utility (Cert2spc.exe), Certificate Manager Utility (certmgr.exe), Certificate Verification Utility (chktrust.exe), Certificate Creation Utility (makecert.exe), Permissions View Utility (permview.exe), PEVerify Utility (peverify.exe), Secutil Utility (SecUtil.exe), Set Registry Utility (setreg.exe), File Signing Utility (signcode.exe) and Isolated Storage Utility (storeadm.exe)[13].

However there are functions in C# that despite all security features and tools that the language provides are still unsafe. A good example would be the function *GetTempPathA.* This function gets the path for the directory where temporary files are stored. The system path information is constantly sought after by attackers or malicious users profiling a target application or system. Path information alone can potentially identify the underlying operating system, installed applications, configurations, and in some cases user and security information[14].

In order to prevent attackers to obtain this kind of information the programmer has to follow the recommended guidelines for programming in C#. In this case ensuring that non-alphanumeric characters are removed from the string before it is processed and that the information is only processed internally by the application would solve the issue. Also a good idea would be to limit the end user’s ability to ascertain or traverse path information.

Other examples of functions that may cause security holes are *ImpersonateDdeClieantWindow, Istrcpyn, OemToAnsiBuff (Class Member) and OemToCharBuffW.* The security leaks created by these functions can be avoided by following the secure programming guidelines.

Perl

Perl was designed to make it easy to program securely even when running with extra privileges. It is straightforward and self-contained[16]. Unlike most command line shells, which are based on multiple substitution passes on each line of the script, Perl uses a more conventional evaluation scheme with fewer hidden snags. Furthermore, because the language has more *builtin* functionality, it can rely less upon external (and possibly untrustworthy) programs to accomplish its purposes[15].

Perl instantly enables a set of special security checks, called *taint mode,* when it detects its program running with differing real and effective user or group IDs[16]. In this mode, Perl takes special precautions called *taint checks* to prevent both obvious and subtle traps. Some of these checks are simple, such as verifying that path directories aren't writable by others. Other checks are supported by the language itself, and it is these checks especially that contribute to making a set-id Perl program more secure for instance than the corresponding C program[15].

However there are commands in Perl that are insecure and the impact on the application can be very high if attackers manage to take advantage of the vulnerabilities. Examples in this sense are *chmod, chown, chroot, exec, fcntl or system*[14]. When using these commands, developers should take into consideration suggested secure programming guidelines in order not to leave open back-doors for attackers.

Spark

One of the most secure programming languages is Spark. The Spade Ada Kernel (Spark) is a language designed for constructing high-integrity systems[17].Spark's design goals are: Logical soundness, Simplicity of formal description, Expressive power, Security, and Verifiability.

Spark has roots in the security community. Back in the 1970s, research into information flow in programs resulted in Spade Pascal and Spark. Spark is widely used today in safety-critical systems, but it is very suited for developing secure software too[17]. It offers complete data- and information-flow analysis. These analyses make it impossible for a Spark program to contain a dataflow error (for instance the use of uninitialized variable), a common implementation error that can cause subtle (and possibly covert) security flaws.

Spark is amenable to the static analysis of timing and usage of memory. This problem is known to the real-time community, where analysis of worst-case execution time is often required[17]. Using Spark's features in this sense, programs can protect against timing-analysis attacks.

All in all Spark is not a "magic bullet", but it has a significant track record of success in the implementation of high-integrity systems[17].

JavaScript

JavaScript was designed as an open scripting language. It is not intended to replace proper security measures, and should never be used in place of proper encryption. JavaScript has its own security model, but this is not designed to protect the Web site owner or the data passed between the browser and the server. The security model is designed to protect the user from malicious Web sites, and as a result, it enforces strict limits on what the page author is allowed to do. They may have control over their own page inside the browser, but that is where their abilities end.

Python

Another scripting programming language is Python. The Python Software Foundation and the Python developer community state that they take security vulnerabilities very seriously. A Python Security Response Team has been formed that does triage on all reported vulnerabilities and recommends appropriate countermeasures.

The language itself contains security features like "rexec" and "Bastion". However vulnerabilities were found in Python too. For instance some of the found and fixed vulnerabilities are *PSF-2005-001 Feb 3, 2005: SimpleXMLRPCServer* vulnerability and *PSF-2006-001 Oct 12, 2006: repr() of unicode strings in wide unicode builds* vulnerability[18].

Conclusion

"Modern research and development has produced various language-level supports for secure systems design. Safe languages provide a flexible and reliable foundation on which to build."[4]

This statement is true but unfortunately nothing in life is completely safe; programming languages are no exception. Several specific security problems have been discovered and fixed in most of the programming languages but there will be always someone that will be able to discover new vulnerabilities in a programming language.

Following the secure recommended programming guidelines in the used programming language and applying the principle of "Correctness by construction" when writing the code for a well designed software product will reduce the risk of vulnerabilities leaks in the final product. In this way the software can be secure, at least for that moment.

Bibliography

1. Sommerville, Ian. Software Engineering. New York: Addison-Wesley Longman, Incorporated, 2006 Chs. 1 2
2. Xavier, Leroy. Computer security from a programming language and static analysis perspective, 2003
3. Ross Anderson and Roger Needham. Programming Satan's computer. In Computer Science Today: Recent Trends and Developments, number 1000 in Lecture Notes in Computer Science, pages 426–441. Springer-Verlag, 1995.
4. Shalka, Christian. Programming Languages and Systems Security, 2005
5. Seacord, Robert C. Secure Coding in C and C++. New York: Addison Wesley Professional, 2005, (Chs 1, 2)
6. McGraw, Gary. Software Security : Building Security In. New York: Addison Wesley Professional, 2006, (Chs. 3, 5, 9, 11)
7. Newsham, Tim. Format String Attacks, 2000.
8. ISO/IEC TR 24731-1:2007 - http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=38841
9. Sun Java JRE GIF Image Processing Buffer Overflow Vulnerability - http://secunia.com/advisories/23757/
10. Java™ Security Overview - http://java.sun.com/javase/6/docs/technotes/guides/security/overview/jsoverview.html
11. Lesson: Security Features Overview http://java.sun.com/docs/books/tutorial/security/overview/index.html
12. Secure Coding Guidelines for the Java Programming Language, version 2.0 - http://java.sun.com/security/seccodeguide.html
13. Security Features in C# - http://www.csharphelp.com/archives/archive189.html
14. Foster, James, and Steven C. Foster. Programmer's Ultimate Security DeskRef : Your Programming Security Encyclopedia. New York: Syngress P, 2004, Pages 23-263, 335-377, 447-469.
15. Perl security - http://search.cpan.org/dist/perl/pod/perlsec.pod

16. Wall, Larry, Tom Christiansen, and Jon Orwant. Programming Perl. Danbury: O'Reilly Media, Incorporated, 2000.
17. Hall, Anthony, and Chapman, Roderick. Correctness by Construction: Developing a Commercial Secure System, 2002.
18. Python Security Advisories - http://www.python.org/news/security/