

Implications of Programming Language Selection On the Construction of Secure Software Systems

Craig E. Ward

Advisor: Ray Toal

December 13, 2004

Department of Electrical Engineering and Computer Science
College of Science and Engineering
Loyola Marymount University

Acknowledgements

I wish to thank my wife, Karin, for all of the support she has given me and for proof reading my work. I also wish to thank Ray Toal for reviewing my work. His suggestions, and especially the questions he asked about the research, greatly enhanced the quality of the result. I also wish to thank Stephanie August for recommending to the University that I be admitted into the graduate program.

Table of Contents

1.	INTRODUCTION	1
1.1.	LANGUAGES ANALYZED	1
1.2.	IDENTIFYING THE VULNERABILITIES	2
2.	ANALYSIS OF VULNERABILITIES	3
2.1.	VULNERABILITY: RACE CONDITIONS	3
2.1.1.	<i>Vulnerability in C</i>	3
2.1.2.	<i>Vulnerability in C++</i>	5
2.1.3.	<i>Vulnerability in Java</i>	5
2.1.4.	<i>Vulnerability in Perl</i>	6
2.1.5.	<i>Vulnerability in ML</i>	6
2.2.	VULNERABILITY: MALICIOUS INPUT	7
2.2.1.	<i>Vulnerability in C</i>	7
2.2.2.	<i>Vulnerability in C++</i>	8
2.2.3.	<i>Vulnerability in Java</i>	8
2.2.4.	<i>Vulnerability in Perl</i>	9
2.2.5.	<i>Vulnerability in ML</i>	11
2.3.	VULNERABILITY: INTEGER OVERFLOW	11
2.3.1.	<i>Vulnerability in C</i>	12
2.3.2.	<i>Vulnerability in C++</i>	15
2.3.3.	<i>Vulnerability in Java</i>	15
2.3.4.	<i>Vulnerability in Perl</i>	16
2.3.5.	<i>Vulnerability in ML</i>	17
2.4.	VULNERABILITY: FORMAT STRINGS	17
2.4.1.	<i>Vulnerability in C</i>	18
2.4.2.	<i>Vulnerability in C++</i>	19
2.4.3.	<i>Vulnerability in Java</i>	20
2.4.4.	<i>Vulnerability in Perl</i>	21
2.4.5.	<i>Vulnerability in ML</i>	22
2.5.	VULNERABILITY: STACK BUFFER OVERFLOW	22
2.5.1.	<i>Vulnerability in C</i>	22
2.5.2.	<i>Vulnerability in C++</i>	23
2.5.3.	<i>Vulnerability in Java</i>	23
2.5.4.	<i>Vulnerability in Perl</i>	24
2.5.5.	<i>Vulnerability in ML</i>	24
2.6.	VULNERABILITY: HEAP BUFFER OVERFLOW	25
2.6.1.	<i>Vulnerability in C</i>	25
2.6.2.	<i>Vulnerability in C++</i>	26
2.6.3.	<i>Vulnerability in Java</i>	27
2.6.4.	<i>Vulnerability in Perl</i>	27
2.6.5.	<i>Vulnerability in ML</i>	27
2.7.	VULNERABILITY: JAVA INNER CLASSES	28
2.7.1.	<i>Vulnerability in Java</i>	28
2.7.2.	<i>Vulnerability in C</i>	29
2.7.3.	<i>Vulnerability in C++</i>	30
2.7.4.	<i>Vulnerability in Perl</i>	30
2.7.5.	<i>Vulnerability in ML</i>	30
2.8.	VULNERABILITY: CLASS COMPARISON BY NAME	31
2.8.1.	<i>Vulnerability in C</i>	31
2.8.2.	<i>Vulnerability in C++</i>	31
2.8.3.	<i>Vulnerability in Java</i>	32

2.8.4.	<i>Vulnerability in Perl</i>	32
2.8.5.	<i>Vulnerability in ML</i>	33
2.9.	VULNERABILITY: C++ VIRTUAL POINTER EXPLOIT	33
2.9.1.	<i>Vulnerability in C++</i>	33
2.9.2.	<i>Vulnerability in C</i>	35
2.10.	VULNERABILITY: POINTER SUBTERFUGE	35
2.10.1.	<i>The StackGuard Canary</i>	35
2.10.2.	<i>The StackShield Return Address Copy</i>	36
2.10.3.	<i>Vulnerability in C</i>	36
2.11.	VULNERABILITY: ARC INJECTION.....	37
2.11.1.	<i>Vulnerability in C</i>	38
3.	CONCLUSIONS	39
3.1.	PROGRAMMING LANGUAGES AS TOOLS	39
3.2.	CHANGE IS A CONSTANT	41
3.3.	SUGGESTIONS FOR FUTURE RESEARCH	41
4.	REFERENCES	41

Table of Tables

Table 1: Languages and versions used for research.....	2
Table 2: Vulnerabilities analyzed.	2
Table 3: Summary of results.	40

Table of Figures

Figure 1: Race condition example written in C.	4
Figure 2: Race condition example in Java.	5
Figure 3: Race condition example in Perl.	6
Figure 4: Race condition example in ML.	7
Figure 5: Malicious input example for C.	8
Figure 6: Sample output from example in C.	8
Figure 7: Malicious input example for Java.	9
Figure 8: Output of example Java program.	9
Figure 9: Example program for Perl along with output.	10
Figure 10: Perl program illustrating <i>taint</i> mode and filtering input.	10
Figure 11: Output of example program in Perl.	11
Figure 12: Moscow ML program illustrating malicious input issue.	11
Figure 13: Loss of precision example for C.	12
Figure 14: Output of program in Figure 13.	12
Figure 15: C program illustrating an overflow from calculation.	13
Figure 16: Output of program in Figure 15.	13
Figure 17: Example of integer overrun that changes the sign value.	13
Figure 18: Sample output from program in Figure 17.	14
Figure 19: Example of an exploit of integer overflow in C.	14
Figure 20: Sample output of program in Figure 19.	15
Figure 21: Example of integer overflow in Java.	15
Figure 22: Sample run of the program in Figure 21.	16
Figure 23: Perl program testing integer overflow.	16
Figure 24: A sample run of the program in Figure 23.	17
Figure 25: Sample ML program showing an integer overflow exception.	17
Figure 26: Format string vulnerability in C.	18
Figure 27: Output of example program from Figure 26.	18
Figure 28: Output of C program used with "%n" specifier.	19
Figure 29: Example program in C++ illustrating alternatives to format strings.	19
Figure 30: Output of example code in Figure 29.	19
Figure 31: Alternate example and sample run for C++.	20
Figure 32: Example code in Java.	20
Figure 33: Run of example code in Figure 32.	21
Figure 34: Perl program and output illustrating format strings.	21
Figure 35: Alternate example program for Perl and output.	22
Figure 36: ML program illustrating immunity to format string vulnerabilities and output.	22
Figure 37: Example program in C illustrating a stack buffer overflow.	23

Figure 38: Example program in Java illustrating immunity to stack buffer overflow problems..	24
Figure 39: Example program in Perl illustrating immunity to stack buffer overflows.....	24
Figure 40: Example ML program illustrating immunity to stack buffer overflow problems.....	25
Figure 41: Example program in C illustrating heap buffer overflow.	26
Figure 42: Sample run of code in Figure 41.	26
Figure 43: Perl program illustrating immunity to heap buffer overflows.....	27
Figure 44: Example output of Perl code in Figure 43.....	27
Figure 45: Example of the Java inner classes issue.	28
Figure 46: Disassembler output outer class from program in Figure 45.	29
Figure 47: Disassembler output for inner class of program in Figure 45.	29
Figure 48: Code fragments illustrating problem of class comparison by name in Java.	32
Figure 49: Perl code fragments illustrating class comparison by name.....	32
Figure 50: Sample code showing C++ VPTR exploit.	34
Figure 51: Program illustrating the pointer subterfuge attack.	36
Figure 52: Diagram illustrating basic return-to-libc exploit.	38
Figure 53: Pattern indicating a program vulnerable to an arc injection attack.	38
Figure 53: Diagram showing how arc injection would effect the stack.	39

Abstract

Selecting an implementation programming language for a software system is one of the most important decisions made during the creation of any software system. Different programming languages have different implications for the difficulty or ease of developing secure code. This paper takes eleven vulnerabilities including buffer overruns, malicious input, and race conditions known to exist in deployed systems and analyzes five different languages (C, C++, Java, Perl, and ML) to compare how each language either aids or hinders the creation of a secure software system.

1. Introduction

Software systems play an increasingly large role in society. The range of software systems spans the spectrum from fun and games to mission and life critical applications. Software is everywhere.

Building these systems is difficult. Designing and building a system that will function as its creators intend is hard. Designing and building a system that will satisfy its users' needs is harder. Additionally, the complexity of the interactions between systems and users creates a new area of concern. This is the area of software security.

Many resources and tools are required when building a software system. A programming language is one of the key elements of that mix. The programming language chosen for a system has a direct effect on how a system is to be created and what means must be used to ensure not only that the system functions as expected, but that it is secure and does not function in unexpected ways.

This paper explores the issue of system security from the standpoint of some popular programming languages. What additional pitfalls come with the choice of a particular language? What benefits can a system creator gain by choosing another language?

1.1. Languages Analyzed

The five languages analyzed for each vulnerability cover a broad range of styles and approaches to programming. Four of the five languages are imperative programming languages, three of

which also support object orientation. The fifth programming language is a functional language. One language is interpreted and one compiles into byte codes for a virtual machine. Selecting a broad variety of programming languages makes the comparisons more meaningful. Each language can be viewed as a representative language for a more general type. Scott [1] provides the categorization of the selected programming languages. Table 1 lists the languages, versions, and platforms used for the research.

Table 1: Languages and versions used for research.

Language	Version	Platform
Java	1.4.2	Mac OS X
C	GCC 3.3	Mac OS X, Cygwin
C++	GCC 3.3	Mac OS X
Perl	5.8	Mac OS X
Standard ML	Moscow ML 2.01	Windows XP, Mac OS X

Sources used to check the details of each language were as follows. For Java, [2] and [3], [4] for C and [5] for C++, [6] for Perl and lastly, [7] for Standard ML.

1.2. Identifying the Vulnerabilities

Vulnerabilities were selected after reviewing the literature on secure programming standards. Two general texts are [8] and [9]. The former is concerned with writing secure software and the latter addresses the issue of how to break software. These sources primarily use C and C++ for their examples with a few in Java and Perl used to illustrate particular points.

Many lists of what to do and what not to do can be found online. Sun Microsystems provides guidelines for both Java and C in [10] and [11]. Additional recommendations for Java are at [12].

Security issues for Perl are covered in [6].

Some of the sources used for identifying and describing the vulnerabilities come from "nontraditional" sources. These are described in [13]. This "parallel world" to traditional conferences and journals is one that cannot be ignored.

Table 2: Vulnerabilities analyzed.

Race conditions	Malicious input
Integer overflow	Format string vulnerabilities
Buffer overflow on the stack	Buffer overflow on the heap
Java inner classes	Class comparison by name
C++ virtual pointer exploit	Pointer subterfuge
Arc Injection	

Just as the selection of programming languages seeks to cover a wide range of types and approaches, the selection of vulnerabilities also seeks to cover more than just a set of problems

with one particular programming language. The vulnerabilities listed in Table 2 range over those that programmers using any language must contend with (race conditions and malicious input) to those that are not only exploits targeted at systems written in a particular language, but particular implementations of that language (arc injection and pointer subterfuge). The hope is that the analysis can have more benefits other than just implying that a language is somehow "not good" or another is "good."

2. Analysis of Vulnerabilities

Each subsection will give a general description of the vulnerability and then follow with examples in each language illustrating how the vulnerability exists or does not exist for that language.

The sequence of the vulnerabilities is significant. The list runs from the most general to the most specific. Some vulnerabilities present security issues regardless of the programming language. Other vulnerabilities are tightly focused only on one implementation of a language.

2.1. Vulnerability: Race Conditions

A *race condition* can occur when a sequence of operations are time dependent in a multithreaded or multiprocessing environment. A thread of instructions will be suspended to allow another thread time to execute. While a thread is suspended, the environment that the thread is depending on can be changed in unexpected ways.

The cause of these problems is usually an implicit, and incorrect, assumption that the operations in question are atomic. The solution to these problems is to isolate the most relevant code in a critical section that can be treated as logically atomic. The general solution is beyond the scope of this paper.

Most race conditions result in faulty behaviors and do not constitute security vulnerabilities. Those that do have security implications relate to file system access [8]. These flaws are called time-of-check, time-of-use flaws, TOCTOU for short. This aspect will be examined in this section.

All of the programming languages in the study contain features that allow code to be written that creates race conditions in file system access.

2.1.1. Vulnerability in C

The primary problem in C comes from using a system call to check whether the current process has rights to that file before actually opening the file. A process using that approach can be suspended between the time the access check is made and the time the actual file is opened. While the process is suspended, another process can change which file is associated with the name. The code in Figure 1 (adapted from [8]) illustrates the problem.

```

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main()
{
    char filename[] = "text.txt";
    FILE *fp;
    if (!(access(filename, (R_OK|W_OK)))) {
        if ((fp = fopen(filename, "r+"))) {
            fprintf(fp, "Write something");
            (void)fclose(fp);
        } else {
            perror("fopen");
        }
    } else {
        perror("access");
    }
    exit(0);
}

```

Figure 1: Race condition example written in C.

The window of vulnerability, the race condition, is the time between the completion of the *access* call and the beginning of the *fopen* call. Because the program is using a name for the file, the actual file the name points to can be changed, especially on file systems that support symbolic links, or another file with malicious content substituted. These are significant issues for scripts run with privileges. (More on the scripting issue below in the subsection covering Perl.)

Not only does this code introduce a race condition, but also it does so by performing an unnecessary step. If the current user does not have the appropriate privileges to write to the file, the access system call will indicate this by its return value, but if the user cannot write to the file, the underlying file system also will cause the fopen library routine to fail. That should be sufficient for an application.

Beyond this simple example, [8] describes how the *passwd* command in some versions of SunOS and HP/UX created a vulnerability by accepting a target password file from the user and then creating temporary copies of the file in order to update it. By using symbolic links to manipulate the meaning of the string given for the target password file, an attacker could get a bogus password entry into any location. (The example illustrated the technique by creating a *.rhosts* file in a targeted user directory. This would allow the attacker to login remotely as that user.)

The primary tactic for countering TOCTOU attacks is to avoid using system and library calls that take a filename as an argument. Use instead the calls that take a file handle or file descriptor. Once the operating system has assigned a file handle or descriptor, it cannot be changed as easily as the manipulation of a filename with symbolic links.¹

¹ Viega and Mcraw [8] also provide a recommended sequence that reliably checks that the file in question does not change during the operation. This sequence is Unix-centric and goes beyond the issue of language itself.

2.1.2. Vulnerability in C++

The same system and library calls available to C are also available to C++. The same issues and remedies apply.

C++ also has its own I/O stream classes. Using these classes and avoiding the C library calls would limit the vulnerability. Once the initial stream object is associated with a filename, the natural tendency would be to use the class object for I/O purposes. This would implicitly be using file descriptors instead of a filename.

2.1.3. Vulnerability in Java

As with C++, Java uses a class structure to facilitate I/O. The *java.io* package (and the new I/O, *java.nio*, added for Java 1.4) have many classes for tailoring the input and output streams of a program.

Java I/O classes can be used to create a race condition similar to the simple C example above. Figure 2 illustrates a similar program. The *File* object has methods for checking the permissions that the current process has for reading or writing a file along with other features of the file. This object can be created with a filename. If a Java program were to use this class to test for access writes and then use the filename again to actually open the file, it would create a race condition between the time the access was checked and the time the file was actually opened. If, however, the same program used the just created *File* object it would reduce the potential of a TOCTOU attack.

```
import java.io.*;

public class ExampleFileIO{
    public static void main(String[] args) {
        if (args.length != 1) {
            System.out.println("error: need a filename");
            System.exit(1);
        }
        File file = new File(args[0]);
        System.out.println((file.canRead() ? "You may" : "You may not")
            + " read " + file);
        System.out.println((file.canWrite() ? "You may" : "You may not")
            + " write " + file);
        FileWriter fw;
        try {
            //fw = new FileWriter(args[0]); // Invite race condition
            fw = new FileWriter(file); // Better
            fw.write("Write something in Java.\n");
            fw.close();
        } catch (Exception e) {
            System.out.println(e);
        }
        System.exit(0);
    }
}
```

Figure 2: Race condition example in Java.

Because Java is running on a virtual machine, this technique depends on the implementation of that machine. It is still possible that internally the JVM is creating the *FileWriter* object from a string. As with the C example above, it is still better to not do the checking in the application but to just try to create the *FileWriter* and let the JVM internals do the checking.

2.1.4. Vulnerability in Perl

Race conditions must be guarded against in Perl. The behavior described in the example for C can be replicated in Perl. See Figure 3.

```
use warnings;
use strict;
my $file = "pl-text.txt";
if (-r $file) {
    open(FH, "< $file")
        or die "can't open $file for reading: $!";
    my $line = <FH>;
    print "$line\n";
} else {
    print "$file cannot be read by you!\n";
}
```

Figure 3: Race condition example in Perl.

This program checks the status of the file before issuing the open request. Because Perl is an interpreted language, the window of vulnerability for this pattern is larger than in C.

The scripting nature of Perl also shows how easy it is for a chain of trust to break. Old versions of the Unix kernel had a race condition vulnerability when they ran scripts with the *setuid* bit set. (Run as the owner of the file instead of the current user.) Executable scripts in Unix start with the characters *#!/* and when these kernels detected them, they would start a new interpreter process (any interpreter, not just Perl) with the user id set to match that of the file. It would then pass the *name* of the script to the privileged process running the interpreter. The time between the kernel detecting the script and passing the name to the interpreter created a window of vulnerability where an attacker could replace the script with one of her own choosing. Newer kernels now pass a type of file handle to the privileged interpreter process.

2.1.5. Vulnerability in ML

The standard basis library for ML has a structure named *OS.FileSys*. This structure contains functions that can check the accessibility of a file for the current user. As with the examples in the other languages, the time between the test and the use of the file is a window of vulnerability. The code in Figure 4 illustrates.

```

$ mosml
Moscow ML version 2.00 (June 2000)
Enter `quit();' to quit.
- load "OS";
> val it = () : unit
- if OS.FileSys.access("text.txt",[A_READ]) then
  print "You can read the file\n"
else
  print "You cannot read the file\n";
You can read the file
> val it = () : unit
-

```

Figure 4: Race condition example in ML.

2.2. Vulnerability: Malicious Input

The Malicious Input vulnerability comes into play when a program makes use of data input from a user or other external source. This data can contain elements that the programmer did not expect and the program does not handle.

A particularly dangerous activity is a program that uses another program in a child process to perform part of the system functionality. The two distinct programs become part of a chain of trust within the system. If something outside of the expected range of data is passed between the two processes, exploits are possible.

The tactic to counter the problem is to validate and sanitize the input before using it. Most programming languages do not provide any built-in assistance for this problem and those features that can be used are easily circumvented by a lazy programmer.

2.2.1. Vulnerability in C

For a high level language, C is very close to the machine and core operating system. The standard library contains the function *system*. This function will fork a child process and start a shell with the command given in its argument. Nothing in the *system* library function does any checking of the arguments. That is left to the programmer. The following code illustrates a simple use of the *system* library call.

```

/*
 * Example of how NOT to handle user input
 */
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char *argv[])
{
    if (argc != 2) {
        printf("usage: %s command\n", argv[0]);
        exit(1);
    }

    printf("Running command \"%s\" for you!\n", argv[1]);
    exit(system(argv[1]));
}

```

Figure 5: Malicious input example for C.

Here is a sample run of the program. Note that because the *system* library call forks a shell, all of the additional capabilities of a shell are present beyond just the command being run.

```

$ ./example_mi "echo Something Wicked This Way Comes > Wicked"
Running command "echo Something Wicked This Way Comes > Wicked" for you!
$ cat Wicked
Something Wicked This Way Comes
$

```

Figure 6: Sample output from example in C.

The example illustrates that the shell meta character ">" was processed as if it had been used on an interactive command line. Other shell meta-characters could also be used, *e.g.* "cat /etc/passwd | mail badguy@evilhost".

2.2.2. Vulnerability in C++

C++ has the same fundamental capabilities as C. The same system library call is available and the issue is the same. The language provides no protection for the program.

2.2.3. Vulnerability in Java

The Java language environment has many built-in security features. However, most of these security features exist to protect the machine from malicious code. The problem of malicious input is still present. A Java programmer must still consider the dangers of data.

The standard language package includes a *Runtime* object and this object has an *exec* method. This method will run a command and its arguments in a process outside of the Java Virtual Machine. The following program illustrates one use of this method. Daconta [14] was useful for figuring out the mechanics of the Java *Runtime* object.

```

/*
 * Example of how NOT to handle user input
 */
public class RunCommand
{
    public static void main(String[] args) {
        if (args.length != 1) {
            System.out.println("usage: java RunCommand command");
            System.exit(1);
        }
        System.out.println("Starting command " + args[0]);
        Runtime rt = Runtime.getRuntime();
        try {
            Process p = rt.exec(args[0]);
            int exitValue = p.waitFor();
            System.out.println("Process p exited with " + exitValue);
        } catch (Throwable t) {
            t.printStackTrace();
        } finally {
            System.exit(0);
        }
    }
}

```

Figure 7: Malicious input example for Java.

The *Runtime.exec* method creates a process and returns a reference to a *Process* object. The *Process* object has methods to query the exit status or to pause the Java thread while the external process runs.

An added wrinkle for an attacker is that, unlike the C and C++ environments, Java does not fork a shell and setup the standard I/O ports. Java only runs a command and its arguments. (If the Java program needs to read from or write to the command, it must set these up manually.) An attacker could not rely on shell meta characters to launch additional commands.

As written, the RunCommand program does not do much that is visually interesting. It must be interrupted at the keyboard when it is given a command that will wait for input.

```

$ java RunCommand "echo hello"
Starting command echo hello
Process p exited with 0
$ java RunCommand "sh"
Starting command sh
^CProcess p exited with 2
$

```

Figure 8: Output of example Java program.

2.2.4. Vulnerability in Perl

In addition to security features that address the issue of malicious code, Perl has a feature that addresses the problem of malicious data. Perl is the only language in this sample that has such a feature. This is the Perl *taint* mode.

Perl considers data that comes from either the command line or standard files to be tainted, that is, untrustworthy. Certain Perl operations are prohibited when tainted data is used. The taint mode is not automatic; it must be turned on with the "-T" switch. The example in Figure 9 illustrates how tainted data affects the Perl *system* command.

```
use strict;
use warnings;

my $arg;
foreach $arg (@ARGV) {
    system "echo $arg";
}

$ perl example_mi.pl Hello Perl World!
Hello
Perl
World!
$ perl -T example_mi.pl Hello Perl World!
Insecure $ENV{PATH} while running with -T switch at example_mi.pl line 6.
$
```

Figure 9: Example program for Perl along with output.

What Perl is complaining about is that the environment is unknown so using the input data cannot be used safely. The modifications to the code in Figure 10 illustrate untainting. (Code for untainting adapted from [6].)

```
use strict;
use warnings;

# Make the environment safer so that external data can be used
delete $ENV{ENV};
$ENV{PATH} = '/bin:/usr/bin';

my $arg;
foreach $arg (@ARGV) {
    # untaint each word in the argument by filtering it for
    # acceptable characters
    if ($arg =~ /^([-@\w.]+)$/) {
        $arg = $1;
    } else {
        die "Bad data in $arg";
    }
    system "echo $arg";
}
```

Figure 10: Perl program illustrating *taint* mode and filtering input.

This version prints the first two arguments as the other when run without taint mode enabled. The last argument has a character that the regular expression will not accept. Figure 11 shows an example of untainting data.


```
$ perl -T example_mi2.pl Hello Perl World!
Hello
Perl
Bad data in World! at example_mi2.pl line 15.
$
```

Figure 11: Output of example program in Perl.

The Perl taint mode is a powerful feature for aiding in the construction of a secure software system.

2.2.5. Vulnerability in ML

The standard ML library includes facilities similar to C and C++ for interacting with the host operating system. The programmer must do filtering and sanitizing within the program itself; the runtime will not help the way Perl does. Figure 12 illustrates.

```
C:\>mosml
Moscow ML version 2.01 (January 2004)
Enter `quit();' to quit.
- load "OS";
> val it = () : unit
- OS.Process.system "cmd.exe";
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\>exit
> val it = <status> : status
```

Figure 12: Moscow ML program illustrating malicious input issue.

2.3. Vulnerability: Integer Overflow

An integer overflow occurs when a value is stored in a location smaller than that location can hold. A common result is that data is lost without the running program being able to detect it. Depending on the implementation, the overflow can be represented by a modular wrapping around to the lowest value or saturating the integer by keeping it as the maximum possible value.

Integers are interpreted as either signed or unsigned. Unsigned values make use of all of the bits to represent the number while for signed integers the high order bit is used as the sign indicator. Mixing signed and unsigned values in an expression can cause problems.

Another possibility is that data can be lost in some languages if an integer of, for example, thirty-two bits is assigned to an integer intended for a sixteen bit value.

All of these problems can cause unexpected behaviors in a running program. These behaviors can result in security vulnerabilities or an unexpected termination of the program.

2.3.1. Vulnerability in C

The C language has several arithmetic types and allows for implicit conversions between them. This creates a lot of opportunity for integer overflows. These overflows are not detected by the runtime. The value in the variable wraps around to the lowest possible value for the type.

2.3.1.1 Loss of Precision

The programmer is allowed to make implicit assignments in C. These can cause loss of precision in the results. The program in Figure 13 illustrates the issue. (All examples for the vulnerability in C are based on [15].)

```
/* ex1.c - loss of precision
 * phrack 60 article 0x0a
 */
#include <stdio.h>

int main(void){
    int l;
    short s;
    char c;

    l = 0xdeadbeef;
    s = l;
    c = l;

    printf("l = 0x%x (%d bits)\n", l, sizeof(l) * 8);
    printf("s = 0x%x (%d bits)\n", s, sizeof(s) * 8);
    printf("c = 0x%x (%d bits)\n", c, sizeof(c) * 8);

    return 0;
}
```

Figure 13: Loss of precision example for C.

The Figure 14 shows the output of the program in Figure 13.

```
$ ./ex1
l = 0xdeadbeef (32 bits)
s = 0xffffbeef (16 bits)
c = 0xffffffffef (8 bits)
$
```

Figure 14: Output of program in Figure 13.

As the thirty-two-bit integer is assigned to smaller variables, bits are lost.

2.3.1.2 Overflow from Calculation

A calculation can result in the overflow of an integer variable as show in the program in Figure 15.

```

/* ex2.c - an integer overflow
 * phrack 60 article 0x0a
 */
#include <stdio.h>

int main(void){
    unsigned int num = 0xffffffff;

    printf("num is %d bits long\n", sizeof(num) * 8);
    printf("num = 0x%x\n", num);
    printf("num + 1 = 0x%x\n", num + 1);

    return 0;
}

```

Figure 15: C program illustrating an overflow from calculation.

When run, the program in Figure 15 shows how the value in "num" wraps around to 0. Note that "num" is an unsigned value. Output is in Figure 16.

```

$ ./ex2
num is 32 bits long
num = 0xffffffff
num + 1 = 0x0
$

```

Figure 16: Output of program in Figure 15.

2.3.1.3 Changing Sign

A calculation can also change the sign of a signed value as shown in the example in Figure 17.

```

/* ex3.c - change of signedness
 * phrack 60 article 0x0a
 */
#include <stdio.h>
int main(void){
    int l;

    l = 0x7fffffff;

    printf("l = %d (0x%x)\n", l, l);
    printf("l + 1 = %d (0x%x)\n", l + 1, l + 1);

    return 0;
}

```

Figure 17: Example of integer overrun that changes the sign value.

When run, the program wraps the value of "l" back to the negative number, as shown in Figure 18.

```
$ ./ex3
l = 2147483647 (0x7fffffff)
l + 1 = -2147483648 (0x80000000)
$
```

Figure 18: Sample output from program in Figure 17.

2.3.1.4 Exploits

Integer overruns are difficult to exploit. One possibility is to overrun the integer being used to protect a copy or allocation operation. Many standard C library routines do not do bounds checking. A common counter measure is to put these routines within an "if" statement that does the check before allowing the routine to be called. If the integer variable can be overflowed, that check will fail and possibly allow for a buffer overrun attack to follow. Figure 19 contains an example of the issue.

```
/* width1.c - exploiting a trivial widthness bug
 * phrack 60 article 0x0a
 */
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[]){
    unsigned short s;
    int i;
    char buf[80];

    if(argc < 3){
        return -1;
    }

    i = atoi(argv[1]);
    s = i;

    if(s >= 80){                /* [w1] */
        printf("Oh no you don't!\n");
        return -1;
    }

    printf("s = %d\n", s);

    memcpy(buf, argv[2], i);
    buf[i] = '\0';
    printf("%s\n", buf);

    return 0;
}
```

Figure 19: Example of an exploit of integer overflow in C.

Figure 20 contains examples of running the program.

```

$ ./width1 5 hello
s = 5
hello
$ ./width1 88 hello
Oh no you don't!
$ ./width1 65536 hello
s = 0
Segmentation fault
$

```

Figure 20: Sample output of program in Figure 19.

2.3.2. Vulnerability in C++

As in C, C++ allows implicit conversion of arithmetic values. C++ has additional syntax forms that do allow for greater clarity in explicit casts. They do not, however, change the fundamental problem.

2.3.3. Vulnerability in Java

Java has problems similar to those found in C. Integer values wrap around without detection in the program. The problem is lessened considerably, however, by the Java type system and built-in bounds checking. Figure 21 is a translation of an example in C.

```

public class Width1
{
    public static void main(String[] args) {
        short s;
        //int s;
        int i;
        char []buf = new char[80];

        i = Integer.parseInt(args[0]);
        s = (short)i;

        if (s >= 80) {
            System.out.println("Oh no you don't!");
            System.exit(-1);
        }

        System.out.println("s = " + s);

        for (i = 0; i < s; i++) {
            buf[i] = args[1].charAt(i);
        }
        System.out.println(buf);
    }
}

```

Figure 21: Example of integer overflow in Java.

The line "s = (short)i;" illustrates one of the defenses provided by Java. The programmer must explicitly force the conversion into a short integer. Java does not allow this to happen implicitly the way C does.

Using the explicit cast allows for the loss of precision and therefore the comparison of the value in "s" to 80 does not protect the copy. However, the copy will not happen because "s" will start out less than "i". This could cause other unexpected results later in the execution of some program using this construct. Output of the program in Figure 20 is in Figure 21.

```
$ java Width1 5 hello
s = 5
hello
$ java Width1 65536 hello
s = 0

$ java Width1 6 hello
s = 6
Exception in thread "main" java.lang.StringIndexOutOfBoundsException: String
index out of range: 5
    at java.lang.String.charAt(String.java:444)
    at Width1.main(Width1.java:28)
$
```

Figure 22: Sample run of the program in Figure 21.

2.3.4. Vulnerability in Perl

The Perl language does not have explicit integers. Perl has a scalar type and the meaning of the scalar is determined dynamically. Perl allows the programmer flexibility without the dangers of integer overflows found in C, C++ and Java. Figure 23 contains a simple example.

```
use strict;
use warnings;

my $l;
my $s;
my $c;

$l = 0xdeadbeef;
$s = $l;
$c = $l;

print("l = ", $l, "\n");
print("s = ", $s, "\n");
print("c = ", $c, "\n");

$l = -0x7fffffff;
$s = $l;
$c = $l;

print("l = ", $l, "\n");
print("s = ", $s, "\n");
print("c = ", $c, "\n");
```

Figure 23: Perl program testing integer overflow.

Figure 24 contains the output from the sample program in Figure 23.

```
$ perl ex1a.pl
l = 3735928559
s = 3735928559
c = 3735928559
l = -2147483647
s = -2147483647
c = -2147483647
$
```

Figure 24: A sample run of the program in Figure 23.

2.3.5. Vulnerability in ML

ML only has the one integer type and it cannot be implicitly converted into another type. The implementation used for this paper does have a limit on the size of an integer value. Unlike the other languages, however, it will tell the user when an overflow happens, as shown in Figure 25.

```
>mosml
Moscow ML version 2.01 (January 2004)
Enter `quit();' to quit.
- val l = 0x3fffffff;
> val l = 1073741823 : int
- val x = l + 1;
! Uncaught exception:
! Overflow
- quit();
>
```

Figure 25: Sample ML program showing an integer overflow exception.

Throwing an exception limits the possibilities of an exploit.

2.4. Vulnerability: Format Strings

This vulnerability grows out of the formatted output features of C. The C standard library function *printf*² is passed a set of arguments that are processed and inserted as characters into an output stream. The first of these arguments is a format string of conversion specifiers that describes the number and type of arguments to follow. These format strings represent a language-within-the-language.

The issue arises when the format string is passed to a *printf* function instead of being hard coded. This allows a possibly malicious caller to control how the *printf* call performs. Giving away this control allows an attacker to more easily reverse engineer a program and find more vulnerabilities.

Fortunately, the problem is easily found by static source code analyzers. According to [13], the vulnerability was largely eliminated shortly after it came to light in the late 1990s. [9, p. 317] It

² The library also has *fprintf* and *sprintf*, which share the same format string language. Formatted input functions have a similar format string language.

is still worth examining, however, as prevention to its accidental re-introduction into new systems.

2.4.1. Vulnerability in C

The code, adapted from [9], in Figure 26 illustrates the issue.

```
/*
 * format string vulnerability
 * adapted from [Hoglund, McGraw, 2004]
 */
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
    if (argc < 2) {
        printf("usage: %s <format-string>\n", argv[0]);
        exit(1);
    }
    printf(argv[1]); /* format string implicit in argument */
    printf("\n");
    exit(0);
}
```

Figure 26: Format string vulnerability in C.

The program is allowing outside input to control the interpretation of the arguments to the printf function. Several features of the format string language allow unsafe behaviors.

2.4.1.1 Printing Data from the Stack

Using the conversion specifier "%08x" can cause printf to print a double word from the stack. Each occurrence of this in the format string will cause the function to look at the next address for the argument to print. Because the format string is an argument, it is on the stack and this allows the attacker to work back up the stack, revealing its contents. By seeing what else is on the stack, an attacker can often gain insight into what to look for next.

Another method is to use the "%s" format to print the stack as a string. This will take everything up to a null byte.

```
$ ./example_fs1 AAAAAA%08x%08x%08x%08x
AAAAAAbffffbc8bffffbd4bffffc408fe50784
$ ./example_fs1 AAAAAA%08x%08x%08x%08x_%s_
AAAAAAbffffbc4bffffbd0bffffc3c8fe50784_???(p??8??) (??P??*(??p??+(?Bà??, (?B†
??, P??0-0???-??_
$
```

Figure 27: Output of example program from Figure 26.

These techniques are useful for probing software and are tactics for reverse engineering of the code. These probes might reveal security or control data that the developers had considered hidden. With this data revealed, an attacker can gain an advantage and potentially access to whatever data or resources the hidden security element was intended to protect.

2.4.1.2 Writing Data

Other conversion specifiers can be used to write data into arbitrary areas of memory. The specifier "%n" will write the number of characters printed up to that point into the address specified by the corresponding argument. Using a field width with the "%u" specifier can add arbitrary padding to this data. This combination can write anything to any address.

```
$ ./example_fs1 AAAAAAx04xF0xFDxF7x41x41x41x41x41%08x%08x%08x%08x%16u%n
AAAAAAx04xF0xFDxF7x41x41x41x41x41bffffba8bffffbb4bffffc208fe50784
2414151164
$
```

Figure 28: Output of C program used with "%n" specifier.

The ability to write to any address allows an attacker to do things like change access flags or the flow of the program.

2.4.2. Vulnerability in C++

The same functions that can cause problems in C are also available in C++. However, C++ has a new, object oriented, set of I/O functions. While C++ programmers can still chose to use the C standard I/O routines, their use is discouraged.

```
#include <iostream>

int main(int argc, char *argv[])
{
    if (argc < 2) {
        std::cout << "usage: " << argv[0] << "<format-string>" << std::endl;
        exit(1);
    }
    std::cout << argv[1] << std::endl;
    exit(0);
}
```

Figure 29: Example program in C++ illustrating alternatives to format strings.

When C++ streams are used, as in Figure 29, the format strings vulnerability disappears.

```
$ ./example_fs2 AAAAAA%08x%08x%08x%08x
AAAAAA%08x%08x%08x%08x
$ ./example_fs2 AAAAAA%08x%08x%08x%08x_%s_
AAAAAA%08x%08x%08x%08x_%s_
$
```

Figure 30: Output of example code in Figure 29.

C++ I/O makes use of an inheritance class structure to standardize the use of output routines. For basic text output, the "<<" operator is overloaded to seamlessly take characters, strings, integers, and other types in a stream of operations. The code shown in Figure 31 illustrates this feature.

```
#include <iostream>

int main(int argc, char *argv[])
{
```

```

    if (argc < 2) {
        std::cout << "usage: " << argv[0] << " argument1 [argumentN]" <<
std::endl;
        exit(1);
    }
    for (int i = argc-1; i > 0; i--) {
        // notice that overloaded "<<" operator takes characters, integers, and
        // strings.
        std::cout << '[' << i << "]" << argv[i] << std::endl;
    }
    exit(0);
}

$ ./example_fs4 ABCDEFG 123456 uwxyz
[3] uwxyz
[2] 123456
[1] ABCDEFG
$

```

Figure 31: Alternate example and sample run for C++.

2.4.3. Vulnerability in Java

Java has some format strings, but they are not as general as the format strings in C *printf* functions. Java format strings are usually used for localization features. The running program can receive indicators from the environment that control how numbers or dates are displayed, *e.g.* using a European date format instead of a US format.³ These format strings do not have the vulnerability.³

Most output in Java is similar to C++ in use, but different internally. In C++, the "<<" operator is an overloaded function that is returning the stream for the next operator to use. In Java, a string concatenation operator, "+", builds strings that are passed to the output stream. For objects that are not strings, the *println* method calls the objects *toString* method automatically.

```

public class ReverseCountEcho
{
    public static void main (String[] args) {
        for (int i = args.length - 1; i >= 0; i--) {
            System.out.println "[" + i + "]" + args[i]);
        }
    }
}

```

Figure 32: Example code in Java.

When the code in Figure 32 is run, the class simply echoes its arguments in reverse order on a separate line.

```

$ java ReverseCountEcho ABCDEFG 123456 uwxyz
[2] uwxyz

```

³ While this paper was being written, Sun Microsystems released a new version of Java that contains a new Formatter object that uses conversion specifiers similar to those in C. These have not been investigated but is it doubtful that these format string specifiers add the vulnerability to Java.

```
[1] 123456
[0] ABCDEFG
$
```

Figure 33: Run of example code in Figure 32.

2.4.4. Vulnerability in Perl

Perl supports functions similar to the C *printf* family in the standard C library. These functions are implemented differently and do not suffer the same problems as the C counterparts. Using the same format strings as in the example for C show that these strings are simply echoed. If, as should be the case for all production code, warnings are enabled, Perl will send error messages to the standard error output stream. Figure 34 contains an example Perl program and its output.

```
use strict;
use warnings;

my $arg = shift(@ARGV);
printf($arg);
print("\n");

$ perl example_fs2.pl AAAAAAx04xF0xFDxF7x41x41x41x41x41%08x%08x%08x%08x%16u%n
Use of uninitialized value in printf at example_fs2.pl line 6.
Use of uninitialized value in printf at example_fs2.pl line 6.
Use of uninitialized value in printf at example_fs2.pl line 6.
Use of uninitialized value in printf at example_fs2.pl line 6.
Use of uninitialized value in printf at example_fs2.pl line 6.
Modification of a read-only value attempted at example_fs2.pl line 6.
$
```

Figure 34: Perl program and output illustrating format strings.

Notice that Perl complains that the "%08x" specifiers are not paired with actual arguments and that the "%n" is attempting to write to a read-only area.

For most output in Perl, the simpler and more efficient print function is recommended [6, p. 594]. This function also illustrates how Perl can evaluate scalar variables for the most appropriate type depending on the context of use. The code is in Figure 35.

```
use strict;
use warnings;

sub some_func
{
    my $count = 0;
    my $str;
    foreach $str (@_) {
        print("[ $count] $str\n");
        $count++;
    }
}

some_func(@ARGV);

$ perl example_fs.pl ABCDEFG 123456 uxyz
```

```

[0] ABCDEFG
[1] 123456
[2] uwxyz
$

```

Figure 35: Alternate example program for Perl and output.

2.4.5. Vulnerability in ML

ML does not use format strings so the vulnerability is not present.

The way that ML handles formatted output is similar to Java. The string concatenation operator, "^", can be used to build a string for output. The difference is that any string conversion functions are not automatically called. Some example code is shown in Figure 36.

```

C:>mosml
Moscow ML version 2.01 (January 2004)
Enter `quit();' to quit.
- val word1 = "Hello";
> val word1 = "Hello" : string
- val word2 = " functional ";
> val word2 = " functional " : string
- val word3 = "world!\n";
> val word3 = "world!\n" : string
- print (word1 ^ word2 ^ word3);
Hello functional world!
> val it = () : unit
- val year=2004;
> val year = 2004 : int
- load "Int";
> val it = () : unit
- print (word1 ^ " " ^ Int.toString(year) ^ "\n");
Hello 2004
> val it = () : unit
- quit();

```

Figure 36: ML program illustrating immunity to format string vulnerabilities and output.

2.5. Vulnerability: Stack Buffer Overflow

A stack buffer overflow occurs when more data is copied into an automatic variable than the programmer had allowed space for. The new data overwrites what was there and in the case of the stack, this can include other variables and other data used by the microprocessor to coordinate the calling and returning from functions. Usually, this results in the program crashing. However, carefully crafted input can cause the microprocessor to execute alternative code. This alternative code could be a worm that then uses the current system as a stepping-stone to the next target. Or it could be “shellcode,” code that causes an interactive shell to be created that the attacker can then use to whatever purpose he or she desires. The canonical description, referenced by virtually all other descriptions of the vulnerability, is [16].

2.5.1. Vulnerability in C

Features of the C and C++ languages make this a significant problem for systems built with these languages. The languages do not have a notion of the actual size of arrays (whether the array is

of characters, integers or some user defined structure does not matter) but depend on the programmer to ensure that the last byte or element is a NULL. Each byte is checked and if it is not a NULL, the operation continues. The code in Figure 37 illustrates the problem. (This program will crash; it is not an actual exploit.)

```
/*
 * Example of a buffer overflow to smash stack in C.
 * Adapted from Aleph One.
 */
#include <stdio.h>
#include <string.h>

void function(char *str) {
    char buffer[16];
    printf("\nIn function...");
    strcpy(buffer, str);
    printf("Leaving function\n");
}

int main() {
    char large_string[256];
    int i;

    for( i = 0; i < 255; i++)
        large_string[i] = 'A';
    function(large_string);
    return(0);
}
```

Figure 37: Example program in C illustrating a stack buffer overflow.

The call to *strcpy* in *function* continues to write data in buffer until it finds a NULL byte. It does not matter if the buffer it is copying into is smaller than the buffer it is copying out of. In this example, the actions of the *strcpy* overwrite the return address so the program crashes with a Segmentation Fault when it tries to return to *main*.

2.5.2. Vulnerability in C++

For this vulnerability, C++ is the same as C. If the sample program is passed through a C++ compiler, the program crashes for the same reason.

2.5.3. Vulnerability in Java

Unlike C and C++, the Java language includes bounds checking on its buffer operations. It is also running on a virtual machine and the code is not aware of the host stack. This vulnerability does not exist in Java. The program in Figure 38 is a straightforward translation of the C example and illustrates how Java is different.

```

public class Example2 {
    static void function(char [] str) {
        char [] buffer = new char[16];
        int i = 0;
        // Force ArrayIndexOutOfBoundsException
        while (str[i] == 'A') { buffer[i] = str[i]; i++; }
    }
    public static void main(String[] args) {
        char [] large_str = new char[256];
        for (int i = 0; i < 255; i++) { large_str[i] = 'A'; }
        function(large_str);
    }
}

```

Figure 38: Example program in Java illustrating immunity to stack buffer overflow problems.

The program in Figure 38 compiles and runs; however, it will crash with the exception *java.lang.ArrayIndexOutOfBoundsException*.

2.5.4. Vulnerability in Perl

As with Java, Perl runs on a virtual machine. While Perl programs do not generate an index-out-of-bounds exception, the Perl runtime is able to extend a string buffer to the limits of the available memory. The vulnerability does not exist in Perl programs. The (awkward) Perl program in Figure 39 prints out two strings of 255 “A” characters.

```

use strict;
use warnings;

sub function
{
    print ("Function called:", @_, "\n");
    my @buffer = @_;
    print ("My Buffer: ", @buffer, "\n");
}

my @large_str;

for (my $index = 0; $index < 255; $index++) { $large_str[$index] = 'A'; }

function(@large_str);

```

Figure 39: Example program in Perl illustrating immunity to stack buffer overflows.

2.5.5. Vulnerability in ML

This vulnerability does not exist in ML for two reasons. Fundamentally, the language does not make use of the side effects. Rather than declare that a buffer exists with sixteen bytes or another buffer exists for 255 characters, ML makes use of function definitions that return strings (or lists or some other structure) of some length.

The ML Array construct includes the ability to compute by side effect. The package includes functions that can update a cell in an array without recreating the array. As in Java, these functions will throw a *Subscript* exception if called with an index outside of the range of the

array. The record in Figure 40 of an interactive session with ML illustrates. (The responses of the ML shell are in *italics*.)

```
- open Array;  
> type 'a array = 'a array  
...  
- val A = array(16, #"A");  
> val A = <array> : char array  
- val B = array(32, #"B");  
> val B = <array> : char array  
- val i = ref 0;  
> val i = ref 0 : int ref  
- while !i < length(B) do (  
  update(A, !i, sub(B, !i));  
  i := !i + 1);  
! Uncaught exception:  
! Subscript  
-
```

Figure 40: Example ML program illustrating immunity to stack buffer overflow problems.

2.6. Vulnerability: Heap Buffer Overflow

A heap buffer overflow is similar to a stack buffer overflow. The difference is that the overwritten memory is not used for the stack.

The heap is memory dynamically managed at runtime. Memory is allocated and deallocated as needed, usually from a larger pool of memory already requested from the operating system.

How heap memory is managed varies from system to system. A common approach uses meta data near the allocated memory buffer. This meta data allows a heap manager to keep track of which buckets are in use, which are free, and which ones can be consolidated.

By gaining control of dynamic memory, an attacker could change security flags or inject code to alter the functioning of the system or as an alternate means of injecting shellcode on systems where the stack is treated as not executable by the hardware.

2.6.1. Vulnerability in C

The standard C library provides routines for allocating and deallocating memory. The main routine to be concerned with here is *malloc*. The *malloc* function returns a pointer to a buffer of memory of a particular size in bytes.

The routines that aid in the stack buffer overflow vulnerability also come into play for this vulnerability. These routines can be used to overflow heap buffer and change the contents of that memory as illustrated by the following program. The example code in Figure 41 is adapted from [8].

```

/*
 * Adapted from [Viega/McGraw, 2002] Example 7-12
 */
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main(int argc, char **argv)
{
    // int      i;
    char      *str = (char *)malloc(sizeof(char) * 4);
    char      *super_user = (char *)malloc(sizeof(char) * 9);
    printf("Address of str is: %p\n", str);
    printf("Address of super_user is: %p\n", super_user);
    strcpy(super_user, "viega");
    if (argc > 1)
        strcpy(str, argv[1]);
    else
        strcpy(str, "xyz");

    printf("Value in buffer str: %s\nValue in buffer super_user: %s\n",
          str, super_user);
    return(0);
}

```

Figure 41: Example program in C illustrating heap buffer overflow.

Running the code twice illustrates the issue. (See Figure 42.) The first run shows where the allocated buffers are and their default contents. The second shows how to change the default for the variable that is not supposed to vary.

```

$ ./vm_example2A
Address of str is: 0x300140
Address of super_user is: 0x300150
Value in buffer str: xyz
Value in buffer super_user: viega
$ ./vm_example2A xyz.....mcgraw
Address of str is: 0x300140
Address of super_user is: 0x300150
Value in buffer str: xyz.....mcgraw
Value in buffer super_user: mcgraw
$

```

Figure 42: Sample run of code in Figure 41.

If the variable "super_user" were used later in the program to validate privileges for the current user, Mr. McGraw could gain root access.

2.6.2. Vulnerability in C++

Although C++ includes additional means for dynamic memory management, the above code is still legal in a C++ program. The general vulnerability is therefore still present.

2.6.3. Vulnerability in Java

The Java programmer does not have direct control of the memory management of the program. This eliminates the problem for a Java program. Additionally, as was the case for the stack buffer overflow, the Java runtime includes automatic bounds checking. An attempt to overflow a Java string or array would generate an *ArrayIndexOutOfBoundsException* exception.

As noted in [9, p. 291], many Java virtual machines are written in C and could therefore have these problems. This is not, however, a fault of the language itself.

2.6.4. Vulnerability in Perl

Perl does not give the programmer direct control of memory allocation and deallocation therefore, as with Java, this attack tactic is not useful against a Perl program. The example in Figure 43 is patterned after the example used for C and illustrates the result.

```
use strict;
use warnings;

my $str;
my $super_user = "viega";

if (@ARGV) {
    $str = $ARGV[0];
} else {
    $str = "xyz";
}

print("Value in buffer str: $str\nValue in buffer super_user:
$super_user\n");
```

Figure 43: Perl program illustrating immunity to heap buffer overflows.

Running the program (see Figure 44) shows how Perl simply allocates the needed memory for the programmer.

```
$ perl example2A.pl
Value in buffer str: xyz
Value in buffer super_user: viega
$ perl example2A.pl xyz.....mcgraw
Value in buffer str: xyz.....mcgraw
Value in buffer super_user: viega
$
```

Figure 44: Example output of Perl code in Figure 43.

2.6.5. Vulnerability in ML

Memory management in ML follows the pattern seen in Java and Perl. The programmer does not have explicit access to the memory management, which makes the heap overflow tactic of limited value.

The non-imperative style used by ML also limits the usefulness of the tactic. Suppose a function were used to grant extra privileges to a process if the user running it was "viega" and that this

value were bound to the name "super_user." The value of super_user at the time of the definition of the privilege granting function would be the value used regardless of how a later phase of the computation might change it. The environment in place when a function is defined is the one used when it is evaluated later.

2.7. Vulnerability: Java Inner Classes

This vulnerability comes about from the Java implementation of object-oriented programming. The other languages in the study either do not support an object-oriented paradigm or do not have this problem. Disagreement exists over whether the issue of Java inner classes constitutes a real vulnerability or that they are actually a logical part of the object-oriented paradigm.

2.7.1. Vulnerability in Java

The feature of inner classes in Java that causes concern is that the access modifiers for the members of the outer class appear to be changed when an inner class is used. A private member is given a package scope. This change breaks the meaning of the access modifier and enables the possibility of undesirable access to private class members.

The Java language specifies that inner classes be treated just like other members of a class. The problem arises because the Java Virtual Machine (JVM) does not directly support inner, or nested, classes. All classes exist as distinct class files that are loaded at runtime. A way is needed to be able to give the appearance that the inner class is a member of the outer class. The method that the Sun compiler uses is not to actually change the access modifier of members, but to create special, static access methods in the outer class that the inner class may use and to insert a member into the inner class to serve as a reference to the outer class.

The example in Figure 45 shows a class that uses an inner class to manipulate private data.

```
public class Flag
{
    class InnerFlag {
        public void incFlag() { flag++; }
        public void showFlag() {
            System.out.println("The hidden flag is " + flag);
        }
    }
    public Flag(int flag) {
        this.flag = flag * 5;
    }
    private int flag;
}
```

Figure 45: Example of the Java inner classes issue.

The Java disassembler, *javap*, shows how the outer and inner classes are changed to allow the inner class to function as shown in Figure 46.

```
$ javap -private Flag
Compiled from "Flag.java"
```

```

public class Flag extends java.lang.Object{
    private int flag;
    public Flag(int);
    static int access$008(Flag);
    static int access$000(Flag);
}

```

Figure 46: Disassembler output outer class from program in Figure 45.

The byte code for the *Flag* class shows two extra static methods. They are the package scope methods that the *InnerFlag* class will use. (Such methods are added only for members that the inner class uses.) Figure 47 shows the same disassembly for the inner class.

```

$ javap -private Flag\$InnerFlag
Compiled from "Flag.java"
class Flag$InnerFlag extends java.lang.Object{
    private final Flag this$0;
    Flag$InnerFlag(Flag);
    public void incFlag();
    public void showFlag();
}

```

Figure 47: Disassembler output for inner class of program in Figure 45.

In the *InnerFlag* class, an extra member has been added to provide a reference to the instance of the outer class. This reference is the argument passed to the added access methods in the outer *Flag* class.

The private member of the *Flag* class is still private, but the compiler has provided package scope methods to access this private data. Theoretically, another class in the package could use the same method. While the Sun compiler does not allow a call to such a class method, e.g. *Flag.access\$000*, another compiler could be altered to do so. Since the JVM does not directly support inner classes, how would the runtime know the difference?

2.7.1.1 Vulnerability or Orthogonality?

Whether inner classes represent a potential vulnerability or indicate that Java treats class members orthogonally is an open question. The answer would depend on the questioner's point of view.

The access modifiers in Java can be viewed as advisory aids in the development of software and not as aspects of software security. Java programs run without a security manager (without the sandbox) can change access modifiers and the values of the members associated with them using Reflection. This is how many IDEs do their work. A security perspective would want access specifiers enforced consistently. Which point of view is best depends on the conditions under which the question is asked [17].

2.7.2. Vulnerability in C

The C programming language does not support an object-oriented approach. An object-oriented paradigm can be used, but that requires great discipline on the part of the programmer.

2.7.3. Vulnerability in C++

C++ also allows for inner, or nested, classes. However, these classes are not automatically allowed any special access to the methods and fields of the encompassing class. The behaviors of public inner classes in Java can be replicated in C++ if the programmer chooses to do so. An inner class can be made a “friend” of the outer class. Classes in other packages, or namespaces, would also have to be declared as friends.

One point of view on the issue is that C++ is being less orthogonal than Java by not treating all members of a class the same way. A security point of view would appreciate the consistency of the meanings of access modifiers. Which is right depends on the particulars of a situation.

2.7.4. Vulnerability in Perl

Perl is a special case. Objects can be created, manipulated, and used in Perl programs, but object orientation is enforced by the programmer. All members of a class are still accessible through the module symbol table. The Perl runtime does not enforce encapsulation. Programmers are expected to have good manners and not require that the runtime environment act as an enforcer. Information hiding is possible by judicious use of lexical scoping and closures, but this is developer choice and not an imperative of Perl object-oriented programming.

Even without strict adherence to the rules of encapsulation, the features of Perl do make the language amenable to an object orientation.

2.7.5. Vulnerability in ML

ML has a module feature that plays the role that classes and objects have in other languages. ML modules are constructed from structures, signatures, and functors.

Structures can be thought of as simple objects. These objects are built from types, functions, exceptions, and other elements. A structure alone can be used as an object, but ML provides an additional means of encapsulation using signatures.

A signature is a definition of an interface. This interface is the public image that a structure will present to other classes. Classes that use the signature will not know what other elements are present in the structure.

Functors are a mechanism used for building new versions of structures. A functor accepts a structure as input and returns a new, modified structure. Functors fill a role similar to templates in C++.

The signature feature of ML prevents the problem found in the implementation of Java inner classes. Elements of a structure not present in the signature are not visible to other classes. In addition, ML does not have the notion “friend” as in C++; if a different interface were required for some class, an additional signature would need to be created.

2.8. Vulnerability: Class Comparison By Name

This vulnerability is from a family of vulnerabilities termed "mix-and-match" [12]. The problem is that an object that is part of a larger system can be replaced by an attacker and provide behaviors other than what the original developers had planned. Checking an object by using its name instead of using the runtime type information makes a mix-and-match attack easier.

This particular form of the vulnerability is particularly dangerous for programming language environments that attempt to handle mobile code. In the set of languages used for this study, only Java and Perl have direct threats to address. The other languages can have a similar issue if the environment supports some form of dynamic linking. Dynamic libraries are supported in Windows and many Unix systems.

If an environment variable is used for a search path to the dynamic library, that can be reset to have the library loaded from a different location. Environment variables can also be part of a malicious input attack. Vulnerabilities can be combined.

2.8.1. Vulnerability in C

The C language does not directly support the concept of an object. A mix-and-match attack against a system written in C would require that the system use a form of dynamic linking. Shared libraries in HP/UX provide a good illustration of the uses, and possible abuses, of dynamic linking.

HP/UX allows a program to specify a search path for shared libraries. This allows the location of the library to change, for example, from one release of the OS to another, without requiring that the program be rebuilt. This feature could also be used by an attacker. By changing the value of the environment variable used by the program, a different, malicious, version of a shared library could be substituted for a legitimate one. The feature is useful, but must be used with care.

2.8.2. Vulnerability in C++

Although an object-oriented language, C++ does not insist on a single root object the way Java does. C++ does not have an equivalent to the Java Class object that can be retrieved for all objects in the system. Therefore, unlike Java, C++ programs do not have a standard, default way of doing a comparison of a class by name. Any program that needed such a capability would require a custom implementation. A poor implementation might not result from issues relating to the programming language.

Systems written in C++ can make use of any dynamic libraries found in the host environment and have the same set of problems as systems written in C. Using the *dynamic_cast* keyword would not detect that a derived class of the base class was not the expected class, *i.e.* that a new malicious class derived from the same base class had been substituted for the legitimate derived class.

2.8.3. Vulnerability in Java

Every object in Java inherits from a root object called *Object*. This root object contains methods that allow for the retrieval of a *Class* object for each class. This *Class* object contains information about the class, including what type of object it is as well as its name. The problem with using the name is that the object gets to decide what that name is. An object can lie. An object cannot hide its type. By comparing the expected type instead of current name, the system has more control over the objects it uses. Figure 48 is the example from [12].

Wrong way:

```
if(obj.getClass().getName().equals("Foo")) // Wrong!  
// objects class is named Foo  
}else{  
// object's class has some other name  
}
```

Right way:

```
if(a.getClass() == b.getClass()){  
// objects have the same class  
}else{  
// objects have different classes  
}
```

Figure 48: Code fragments illustrating problem of class comparison by name in Java.

2.8.4. Vulnerability in Perl

Object-oriented programming in Perl is based on the Honor System. The Perl runtime does not enforce encapsulation. (Although a programmer can simulate it with judicious use of lexical scoping and closures.) Perl also does not mandate one root object that all objects can have in their inheritance tree. What it does provide is a *UNIVERSAL* object that will always be part of the method lookup process. This object contains similar methods to those employed by Java.

The *UNIVERSAL* object has two methods: *isa* and *can*. The first is passed an object name and the method returns true if the name matches that of the invoking object, that is, the object "is a" type of that name. The latter is passed a name of a method and returns true if that method is part of the object, that is, it "can" call that method name. The code snippets in Figure 49 that illustrate these universal methods are from [6].

```
$fh = FileHandle->new();  
if ($fh->isa("IO::Handle")) {  
    print "\$fh is some sort of IOish object.\n";  
}  
  
if ($invocant->can("copy")) {  
    print "Our invocant can copy.\n";  
}
```

Figure 49: Perl code fragments illustrating class comparison by name.

This may look like Perl is making the same error as Java, but the names in question are not those provided by the object, but are the names stored in the symbol tables used by Perl to keep track of things.

2.8.5. Vulnerability in ML

Although many of the features of an object-oriented language can be created in Standard ML, the language is not formally object-oriented. The issues of the mix-and-match family of vulnerabilities are fundamentally about not using the type information correctly. The ML language is strongly typed and should not have this problem. As with C and C++, however, the underlying environment could allow for substitution of dynamically loaded libraries.

2.9. Vulnerability: C++ Virtual Pointer Exploit

The C++ Virtual Pointer (VPTR) exploit builds upon the basic stack buffer overflow exploit. It is, however, much more narrowly focused than the general exploit. The exploit described in [18] only relates to the GNU C++ implementation for Linux running on the Intel IA32 platform.

As noted above, Java, Perl, and ML are not susceptible to stack buffer overflow attacks. In addition, implementations of these languages do not store objects on the stack, but use the heap. The techniques used by these languages to manage the heap also make these languages immune to this type of attack.

2.9.1. Vulnerability in C++

Object-oriented programming includes the idea of polymorphism. Polymorphism allows a class to reuse a name of a method that exists in one of its parent classes. The appropriate method is determined at runtime by dynamic binding, as opposed to static binding, which can be done at compile time. A C++ program can identify a method as a virtual function by keyword as shown in the sample program in Figure 50 from [18].

```

#include <stdio.h>
#include <string.h>

class BaseClass
{
    private:
        char Buffer[32];
    public:
        void SetBuffer(char *String)
        {
            strcpy(Buffer,String);
        }
        virtual void PrintBuffer()
        {
            printf("%s\n",Buffer);
        }
};

class MyClass1:public BaseClass
{
    public:
        void PrintBuffer()
        {
            printf("MyClass1: ");
            BaseClass::PrintBuffer();
        }
};

class MyClass2:public BaseClass
{
    public:
        void PrintBuffer()
        {
            printf("MyClass2: ");
            BaseClass::PrintBuffer();
        }
};

void main()
{
    BaseClass *Object[2];

    Object[0] = new MyClass1;
    Object[1] = new MyClass2;

    Object[0]->SetBuffer("string1");
    Object[1]->SetBuffer("string2");
    Object[0]->PrintBuffer();
    Object[1]->PrintBuffer();
}

```

Figure 50: Sample code showing C++ VPTR exploit.

The two "MyClass" definitions inherit from a common base class that provides two methods. One is static while the other is marked as "virtual." Each "MyClass" defines a new implementation of the virtual function. When these definitions are used to create objects in the

program, the compiler is able to setup the call to the *SetBuffer* function statically, but it must setup dynamic binding for the calls to *PrintBuffer*. The created objects contain pointers into a table of function pointers. These *virtual pointers* (VPTRs) are used to find the correct method to call.

The vulnerability comes about from the way the GNU C++ compiler implements the dynamic in combination with the faulty usage of the *strcpy* library call in *SetBuffer*. The VPTR is located on the stack above the object and if the call to "SetBuffer" can be passed an arbitrary string, the VPTR could be overwritten to point to code supplied by the attacker.

The exploit requires more conditions than the standard stack buffer overflow and these conditions are likely rare. (The paper by Rix [18] includes additional possibilities that are even more rare.) It is important to note as an illustration that simply using objects does not necessarily imply safe code.

2.9.2. Vulnerability in C

Standard C does not support objects, but it does include pointers to functions and the calling of functions by address. Idioms that take advantage of this could suffer a similar attack.

One approach to selecting one function from a set of possible functions, all with the same signature, is to build a dispatch table that includes a pattern to select by along with the address of the function to call. The code that looks up the pattern and then calls the function is simple and compact. If the dispatch table were located within reach of one of the library calls known to overflow, the address of the function called could be overwritten.

2.10. Vulnerability: Pointer Subterfuge

Some security-after-the-fact techniques are in use to counter the stack buffer overflow vulnerabilities in existing code. Two of these are StackGuard and StackShield. (A third tactic will be address in the section on arc injection.) Each attempts to detect a stack overrun. Unfortunately, while these tactics can provide some basic protection, neither can resist a persistent attack against a vulnerable program.

2.10.1. The StackGuard Canary

StackGuard, according to [19], uses the technique of placing a extra word, called a canary after the old tactic of using a canary to detect toxic gases in mines, next to the return address. The basic stack buffer overflow rewrites the return address of a function by overwriting bytes in the stack so in order to overwrite a canary protected return address, the canary must also be smashed. The function call prolog creates the canary and the epilog checks it. If the check fails, the program is terminated.

Two types of values are used for the canary. The first, a "terminator," is just the four bytes 0x000d0aff. These are a NULL, carriage return, line feed and end-of-file. If an attack attempts to spoof the canary in the overflow, one of these characters should cause a string operation to terminate. The second is termed "random." It is a randomly generated number. This number only

needs to be random enough to make prediction difficult. (It does not need to be cryptographically strong.)

2.10.2. The StackShield Return Address Copy

StackShield takes a different approach. Rather than add to the existing stack format, it copies the return address to a separate stack. The function call epilog compares the return address in the stack with the copy.

Bulba and Kil3r in [19] show how each stack overflow counter measure can be overcome.

2.10.3. Vulnerability in C

The example pointer subterfuge attacks are designed only for a Linux system running on an Intel IA32 platform.

2.10.3.1 Attack Against StackGuard

The authors of [19] use the program in Figure 51 to illustrate their techniques.

```
int f (char ** argv)
{
    int pipa; // useless variable
    char *p;
    char a[30];

    p=a;

    printf ("p=%x\t -- before 1st strcpy\n",p);
    strcpy(p,argv[1]);           // <== vulnerable strcpy()
    printf ("p=%x\t -- after 1st  strcpy\n",p);
    strncpy(p,argv[2],16);
    printf("After second strcpy ;)\n");
}

main (int argc, char ** argv) {
    f(argv);
    execl("back_to_vul","",0);  //<-- The exec that fails
    printf("End of program\n");
}
```

Figure 51: Program illustrating the pointer subterfuge attack.

The program illustrates the characteristics that must exist in a vulnerable program for StackGuard and StackShield protections to fail.

1. A pointer such as *p* must be located next to a buffer such as *a*.
2. A misused library routine that can execute an overflow into *p*. In the example, this is *strcpy*.
3. A second copy function that uses *p* as the address of the buffer to write without *p* having been initialized.

This may seem like an odd set of requirements, but [19] report just such a bug in a version of wu-ftpd.⁴

The tactic is to use an overrun on the pointer p to have it point to the address of the function return address on the stack. Then use the following copy function to overwrite the function return address.

The authors of [19] provide some shellcode that illustrates their exploit under Linux. That code is not reproduced here.

2.10.3.2 Attack Against StackShield

The tactic described above would not work against StackShield. The change in the return address would be detected and program execution halted. However, having demonstrated the ability to write to any particular address, an attack on a StackShield protected program would change the value in the pointer variable p to such that it will overwrite a function address in *fnlist*, the list of functions called when a program terminates by calling the library routine *exit*.

2.10.3.3 Changing libc Addresses

The paper illustrates another way to attack these protections. If the pointer p can point anywhere, why not change the address of the *printf* function to that of the system function? This is a variant of the return-to-libc attack and will be a key part of the arc injection exploit.

2.11. Vulnerability: Arc Injection

Another approach to preventing stack buffer overflow exploits is to make the stack itself not executable. The standard stack buffer overflow flows its malicious code into the stack and changes the return address of the function to point to this injected code. By making the stack non-executable, the general case is foiled. The arc injection technique is a counter to the non-executable stack counter measure. It still requires, however, a vulnerable program that misuses one of the C library functions that do not do bounds checking.

As with the pointer subterfuge attack, the arc injection attack is much more narrowly focused. The description of the attack in [20] applies GNU C (and C++) under Linux on the Intel IA32 platform. The author of the paper, however, claims that the general technique can work on any system that uses the GNU C calling convention and the ELF (Executable and Linking Format) specification for the format of executable files. The exploit can be attempted on Solaris i386 and FreeBSD in addition to Linux.

The arc injection attack is a variation on the stack buffer overflow vulnerability. Because Perl, Java, and ML do not suffer from this vulnerability, programs written in these languages are not at risk of this vulnerability.

⁴ Bulba and Kil3r name the vulnerability the "wu-ftpd 2.5 mapped_path bug."

2.11.1. Vulnerability in C

The arc injection technique builds upon the return-to-libc tactic. The basic idea is that rather than insert the code to execute into the stack and run it there, the stack is manipulated to change the flow of control to call functions, usually the library function *system*, to perform the exploit. The diagram in Figure 52 from [20] of a hypothetical stack illustrates the idea.

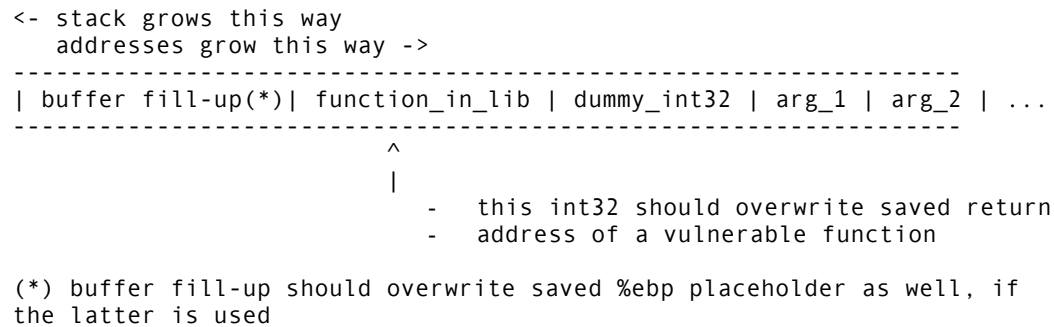


Figure 52: Diagram illustrating basic return-to-libc exploit.

The library call *function_in_lib* is the address that the vulnerable function returns to. When it starts executing, the *dummy_int32* is in the place of the return address for the library function and *arg_1*, and so on, are its arguments. That the program that executes the shell code will crash if the *dummy_int32* value is meaningless is not important to the attacker.

An advance of the technique is to make use of the *dummy_int32* position. Instead of just inserting a dummy address, the position is used to invoke another library routine. These can be chained in order to call more than one library function.

The reason this technique might be needed for a successful attack is that many programs designed to run from a privileged user ID can change the effective ID of the process to one of little or no privileges. The chaining of calls to libc functions can give back the privileges before executing the shellcode. Attackers want to create a shell with root privileges; getting a shell running as "nobody" does not help the attacker much.

2.11.1.1 Finding Injection Points

Crafting an arc injection attack requires detailed knowledge of the calling sequence on the target platform, in this example, Intel IA32. The attacker needs to disassemble the target program and look for patterns in the assembly instructions. Depending on the findings, a different tactic can be used.

Some optimizations by the GNU C compiler (-fomit-frame-pointer) will change how the frame pointer is handled in the calling sequence epilog. The pattern will look something like Figure 53.

```
ep1g: addl    $LOCAL_VARS_SIZE, %esp
      ret
```

Figure 53: Pattern indicating a program vulnerable to an arc injection attack.

This allows for chaining of library calls. The diagram in Figure 53, from [20], illustrates.

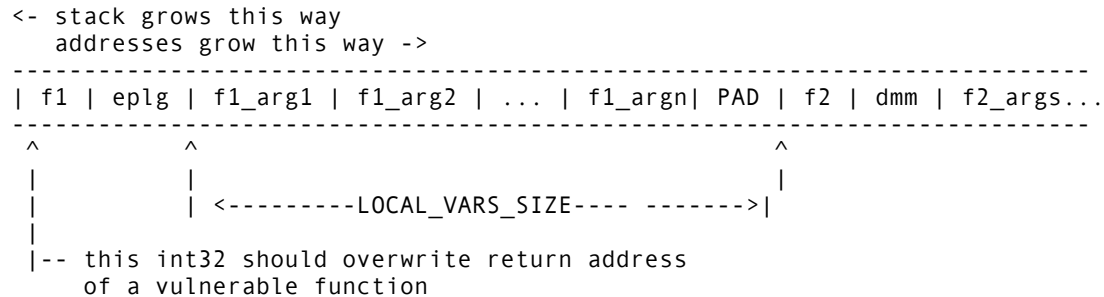


Figure 53: Diagram showing how arc injection would effect the stack.

The "PAD" is some number of non-zero bytes to make the number of bytes come out to match `LOCAL_VARS_SIZE`. When the function returns from *f1*, the *eplg* code changes the stack to point to the beginning of the call to *f2* with its arguments.

The Nergal paper also includes patterns to look for in programs that are not optimized to omit the frame pointer as well as additional cases susceptible to an arc injection attack. The paper also includes extensive sample code illustrating the attack.

The point for the purposes of this paper is not the actual techniques themselves, but that after-the-fact security patches are extremely difficult. The arc injection and pointer subterfuge attacks illustrate just how hard, and how likely to fail, any after-the-fact strategy actually is. A software system created with vulnerabilities remains vulnerable.

3. Conclusions

3.1. Programming Languages as Tools

Different programming languages present different challenges to secure coding, but no programming language is completely secure. Table 3 summarizes the results of the analysis. Each language in the study is at risk for some of the vulnerabilities reviewed. No programming language is at risk from all of the potential vulnerabilities.

Table 3: Summary of results.

Vulnerability/Language	C	C++	Java	Perl	Standard ML
Race conditions	Yes	Yes	Yes	Yes	Yes
Malicious input	Yes	Yes	Yes	Yes	Yes
Integer overflow	Yes	Yes	Yes	No	No*
Format string vulnerabilities	Yes	Yes†	No	No	No
Buffer overflow on the stack	Yes	Yes	No	No	No
Buffer overflow on the heap	Yes	Yes	No	No	No
Java inner classes	No	No	Yes	Yes	No
Class comparison by name‡	No	No	Yes	Yes	No
C++ virtual pointer exploit	No	Yes	No	No	No
Pointer subterfuge	Yes	Yes	No	No	No
Arc Injection	Yes	Yes	No	No	No

*ML throws an exception for integer overflows.

†When using C libraries.

‡Most significant for mobile code but any language that uses dynamic linking can also have a problem.

These results indicate that the security requirements of the system must be considered before the software construction can begin. A system with strong security requirements should be built with a programming language that creates fewer potential problems for the developers. These criteria when combined with other requirements such as the speed of execution or ease of code generation can result in better choices for implementation languages. Any programming language can be misused; any programming language can be used to create insecure software.

A Programming language such as C might appear to be insecure from this analysis; however, this would be an overstatement of the problem. What this analysis means is that, as with any tool, the results of using these programming languages depend on the skill with which they are used. Using C implies additional coding risks that need to be addressed.

An object orientation in the language does not help much with security issues. Some of the risks of using C are mitigated in C++ by using the newer, object oriented I/O classes. But some implementations of C++ bring their own unique hazards as shown by the VPTR attack.

Java programs might execute more slowly than a similar program written in C or C++, but by using a virtual machine that incorporates security conscious features, Java helps the coder write more secure code; although care must be taken if the system uses inner classes.

The taint mode and Safe module features of Perl are a great benefit to a secure system. If the system can accept the interpreted, and thus slower, execution speeds, Perl would be an excellent choice.

In the analysis of these vulnerabilities, Standard ML is the most inherently secure programming language. However, ML is not widely used in industry and the lack of vulnerabilities in such systems could be because not enough people have tried attacking them.

The people who plan and develop software systems can not ignore the security implications of programming language choice. If the choice is to use a language such as C because speed of execution is important, then it is a necessity that additional tools, such as static code analysis tools, be acquired. Extra time to use these tools must be allocated. As shown by the pointer subterfuge and arc injection attacks, trying to graft security onto insecure systems is extremely difficult. The results of this study provide data supporting the argument that code security must be built-in from the beginning.

3.2. Change is a Constant

Programming languages continue to change over time. Sun Microsystems released a new version of Java after work on the paper started. Rather than halt the work on the paper and risk getting bogged down in learning the new features, the research was limited to the version of Java that existed as a stable release at the time of the start of the research. Perl is also rumored to be undergoing a new release cycle soon. These changes in the programming languages could change some of the results.

3.3. Suggestions for Future Research

The list of vulnerabilities used for this paper is not exhaustive. There are others. The analysis could be expanded to include these additional vulnerabilities.

The selection of programming languages is also not exhaustive. Programming languages such as C#, Visual Basic, Python, JavaScript, or PHP could be analyzed for risks associated with these, or other, vulnerabilities.

Finally, code analysis tools, such as [21, 22, 23] are becoming available in academic and open source settings. These could be investigated for their effectiveness in detecting the vulnerabilities mentioned in this paper. We might also wish to research how much effort would be required for an automated tool to detect particular vulnerabilities.

4. References

- [1] M. Scott, *Programming Language Pragmatics*, Morgan Kaufmann, 2000.
- [2] D. Flanagan, *Java in a Nutshell 3rd Edition*, O'Reilly & Associates, 1999.
- [3] B. Eckel, *Thinking in Java 2nd Edition*, Prentice Hall, 2000.
- [4] B. Kernighan and D. Ritchie, *The C Programming Language 2nd Edition*, Prentice Hall, 1998.
- [5] R. Lischner, *C++ in a Nutshell*, O'Reilly & Associates, 2003.
- [6] L. Wall, T. Christiansen, and J. Orwant, *Programming Perl 3rd Edition*, O'Reilly & Associates, 2000.

- [7] J. Ullman. *Elements of ML Programming ML97 Edition*, Prentice Hall, 1998.
- [8] J. Viega and G. McGraw, *Building Secure Software*. Addison-Wesley. 2001.
- [9] G. Hoglund and G. McGraw, *Exploiting Software: How to Break Code*. Addison-Wesley, 2004
- [10] "Security Code Guidelines," Sun Microsystems, Inc.
<http://java.sun.com/security/seccodeguide.html>, 2000.
- [11] R. Teer, "Secure C Programming." Sun Developer Network Community.
<http://developers.sun.com/solaris/articles/secure.html>, 2001.
- [12] G. McGraw and E. Felten, "Twelve rules for developing more secure Java code." Java World. http://www.javaworld.com/javaworld/jw-12-1998/jw-12-securityrules_p.html, 1998.
- [13] J. Pincus and B. Baker. "Beyond Stack Smashing: Recent Advances in Exploiting Buffer Overruns", *IEEE Security & Privacy*, July/August 2004, pp. 20-27.
- [14] M. Daconta, "When Runtime.exec() won't", *Java World*, December 2000,
http://www.javaworld.com/javaworld/jw-12-2000/jw-1229-traps_p.html.
- [15] Blexim, "Basic Integer Overflows", *Phrack*, December 2002,
<http://www.phrack.org/show.php?p=60&a=10>.
- [16] Aleph One, "Smashing the Stack for Fun and Profit", *Phrack*, November 1996,
<http://www.phrack.org/show.php?p=49&a=14>.
- [17] J. Steven, "Re: Java and 'private'", Security Focus email list, November 2, 2000,
<http://www.securityfocus.com/archive/98/143181>.
- [18] Rix, "Smashing C++ VPTRS", *Phrack*, May 2000,
<http://www.phrack.org/show.php?p=56&a=8>.
- [19] Bulba and Kil3r, "Bypassing StackGuard and StackShield", *Phrack*, May 2000,
<http://www.phrack.org/show.php?p=56&a=5>.
- [20] Nergal, "The Advanced Return-to-lib(c) Exploits", *Phrack*, January 2002,
<http://www.phrack.org/show.php?p=58&a=4>.
- [21] J. Viega, J. Bloch, T. Kohno, and G. McGraw, "Token-Based Scanning of Source Code for Security Problems", *ACM Transactions on Information and System Security*, Vol. 5, No. 3, August 2002, Pages 238–261.

[22] D. Gilliam, J. Kelly, J. Powell, M. Bishop, "Reducing Software Security Risk through an Integrated Approach," *NASA Goddard Software Engineering Workshop*, pp. 36-42, November 2001.

[23] D. Wheeler, *Flawfinder*, <http://www.dwheeler.com/flawfinder/>.