Programming Languages Security: A Survey For Practitioners

Nihal A. D'Cunha, Massimiliano Pala, Sean W. Smith
Department of Computer Science
Dartmouth College
nihal,pala,sws@cs.dartmouth.edu

Abstract

Security vulnerabilities, such as buffer overflows, induced by programming errors are the fundamental cause for software insecurity. Secure protocols and technologies have made incredible advances in the past few years while programmers continue to face the same problems and predicaments during application development. This paper focuses on programming languages and tools that attempt to guarantee safety and security. It surveys security vulnerabilities in programming languages, suggests safe language alternatives and reviews the range of software solutions to the security flaws in C in an attempt to provide a complete overview of the existing problems and their possible solutions.

The intended target audiences for this paper are software managers, developers and programmers. The paper addresses issues related to building secure applications and writing secure code by making known what choices are available when starting out on new safety-critical projects.

1 Introduction

In today's networked world where computers have become ubiquitous, computer attacks have become a serious problem. Nearly all applications nowadays need to be secure. However, writing secure code is non-trivial. Security holes in software are common, and the problem is growing. Programmers have to design and write code that is robust and withstands attacks by malicious users. There are numerous exploits and attacks that deliberately seek out and cause exceptional or uncommon situations, such that unwarranted privileges are obtained. Statistics from the CERT Coordination Center at Carnegie Mellon University, reveal that after a marginal decrease in the number of reported software vulnerabilities in 2003 and 2004, reports are on the rise once again [13], as shown in Figure 1.

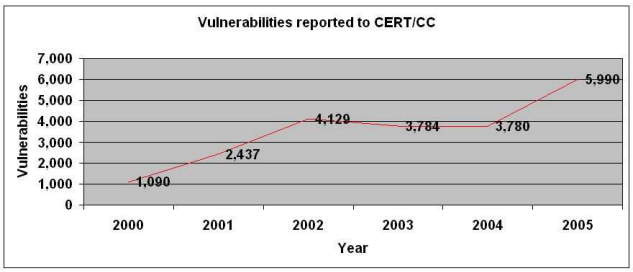

Figure 1: Vulnerabilities reported to CERT/CC 2000-2005

In this paper we survey the issues involved in designing, writing and implementing code that is robust against attacks. We look at potential vulnerabilities, outline possible software solutions and review most relevant safe programming languages.

Since C is the most prevalent language for software development and because most other language implementations make use of C-developed libraries, we take a detailed look at it from a security point of view.

We enumerate vulnerabilities in C that permit several dangerous exploits, making it an unsafe programming language, and outline several tools that help detect and prevent these security flaws in C code.

The rest of the paper is organized as follows. Section 2 is a survey of the common security vulnerabilities in programs. Section 3 outlines a few modern day safe programming languages. Section 4 enumerates tools that can be used to detect and overcome security flaws in C. Finally, Section 5 presents our conclusions.

2 Survey of Vulnerabilities

Memory errors are the most commonly used means by which attackers attempt to gain control over an application [100].

Memory-safety is therefore a prerequisite for security, and attacks that exploit memory-safety are most often to blame for insecurity in deployed software [93]. Attackers exploit these

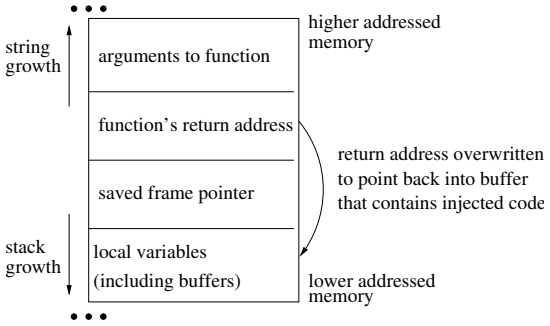

Figure 2: Stack smashing

vulnerabilities, to transfer program control to code that they have injected (which is usually to start a new *shell* with root permissions). A memory-safe program will never overwhelm a buffer or write carelessly over memory that it is not supposed to.

2.1 Buffer Overflows

Buffer overflows are the cause of several software vulnerabilities that lead to security breaches. Most buffer overflow related problems stem from input that is not checked for sanity before use. In this section we analyze Stack-based and Heap-based types of buffer overflow.

Stack-based

A buffer, which is an array of identical datatypes stored in contiguous memory locations, is easy to overflow, as not all programming languages implement bounds checking or maintain array-size information. At run-time, programs are able to write data beyond the end of the array, overwriting adjacent memory areas which usually contain the address to resume execution at after a function has completed execution.

For every function call, a new stack frame is added to the top of the stack. The stack frame contains arguments to the function, the return address, the frame pointer, locally declared variables (including buffers), and other data. When excess data is written into a buffer because proper checking is not performed, the extra data will overflow into the adjacent higher addressed memory. This can be abused to overwrite the return address and hence change the flow control of the program. When the function returns, control will be transfered via the modified return address to a specific address which is usually where the attacker's code is stored, allowing it to execute with the same privileges as the exploited program. Code that does this is said to "smash the stack" [1]. A stack smashing attack is shown in Figure 2.

A *return-to-libc* attack modifies the return address to point to pre-existing functions (e.g., such as those in the `libc` standard library) without the need to inject malicious code into the program.

Buffer overflows can also be used to overwrite security sensitive variables or control data stored in memory areas adjacent to the buffer being overflown.

Other stacked-based exploits involve frame pointer overwriting [49] and indirect pointer overwriting [10].

Heap-based

Dynamically allocated variables (those allocated by the `malloc()` family of functions) are created on the heap segment at run-time by the application. Heap-based arrays are overflown in a manner similar to that of stack-based arrays, except that the heap grows from lower addressed memory to higher addressed memory while the stack grows from higher addressed memory to lower addressed memory. As there are no return addresses stored on the heap, alternative methods are used to manipulate the control-flow.

Heap memory is usually divided and allocated in *chunks* which contain memory management information within them. When a heap buffer overflow occurs, it attempts to overwrite the memory management information of the next chunk in memory with specific values in order to transfer control to the attack code.

Other techniques of overflowing the heap are either by overwriting function pointers [16] or virtual function pointers [75].

2.2 Dangling Pointer Errors

A *dangling pointer* is a reference to an object that no longer exists at that address. Dangling pointers could arise from various conditions [57]:

- reference to an object is retained after explicit deallocation of it on calling `free()`.
- reference to a stack-allocated object is retained after the relevant stack frame has been popped.
- reference to a heap-allocated object is retained after the relevant block has been freed and reused.

Conventionally, C compilers do not verify pointer deferences, resulting in dangerous and elusive dangling pointer dereference vulnerabilities. Programs that deference dangling pointers usually crash, produce garbled output or display unpredictable behavior. Although, a special case of the dangling pointer reference bug called the *double free vulnerability* could cause memory corruption resulting in an attack [41]. A double free attack could occur when a program attempts to free memory that has already been previously deallocated [22].

```
free(x);
/* code */
free(x);
```

When a program calls `free()` twice with the same argument, the heap's memory management data structures become corrupted [29]. This corruption can be exploited by an attacker to execute arbitrary code using the privileges of the exploited application, leading to a partial or total compromise of the system.

2.3 Format String Bugs

Format string bugs are caused by unfiltered user input that is passed as the format string argument to specific C formatting functions, such as those of the `printf()` family of functions.

Format strings use format specifiers, such as %s, %x, and %n, to indicate to the compiler the format of the output that the function should produce. Format functions retrieve arguments for the format specifiers off of the stack. For example, `printf(''%s'', buf)`, here the string `buf` will be popped off the stack. However if this statement is carelessly written as `printf(buf)`, then `buf` will be interpreted as a format string, and will be parsed for any format specifiers it might have. An attacker can take advantage of this and specify a carefully crafted format string to the format function to control what the function pops from the stack [100].

By using the %n formatting specifier, an attacker can cause `printf()` to write the number of bytes printed so far into a location specified by a pointer argument (`int *`); it can be used to write arbitrary values to arbitrary locations chosen by the attacker.

Format string vulnerabilities can also lead to denial of service attacks by employing numerous instances of the %s format specifier to read data off the stack until the program tries to read from an illegal address (i.e. an unmapped address), which will cause it to crash.

Format string bugs originate because of C's type-unsafe argument passing mechanisms. Neither the type nor the count of arguments passed are checked at run-time or compile-time. It is the responsibility of the function taking on the arguments to pop the appropriate number, type, and order of arguments off of the stack [17].

2.4 Integer Inaccuracies

Integer inaccuracies [5] fall into two classes: *integer overflows* and *integer signedness* errors.

Integer overflow errors occur when an integer either becomes greater than its datatype's maximum value or smaller than its minimum value. If one tries to put a value into a datatype that is too small to hold it, the high-order bits are dropped and only the low-order bits are stored. That is, modulo-arithmetic is performed on the value before storing it to make sure it fits within the datatype. Most compilers tend to disregard this overflow, causing programs that do not anticipate this unexpected or inaccurate result to potentially fail.

If memory is being allocated based on an unsigned integer datatype's value and the value wraps around, insufficient memory might be allocated leading to a possible heap overflow.

Integer signedness errors occur when an unsigned variable is treated as signed, or when a signed variable is treated as unsigned. Computers internally do not distinguish between the way signed and unsigned variables are stored leading to this type of insidious bug.

If a programmer passes a signed negative integer as an argument to a function expecting an unsigned value, such as `memcpy()`, the signed negative integer will bypass any size checks and be implicitly cast to an unsigned integer. This cast will cause the value passed to wrap around and become a large unsigned positive value. If this value is now used as the length of bytes that `memcpy()` has to copy from source to destination it will result in `memcpy()` copying well past the end of the destination buffer resulting in a buffer overflow.

Bugs could also arise if an integer overflows and wraps around to a negative number. For example, the addition of two large signed positive integers (say, `s1` and `s2`) could wrap around to form a negative signed integer. This negative signed integer would pass any maximum size checks but when the individual values (i.e. `s1` and `s2`) are used they could be large enough to write past the end of buffers causing a buffer overflow.

2.5 Type-cast Mismatches

Type-casting refers to converting a variable of one datatype into another type. Conversion can be done either implicitly or explicitly. Type-casting is risky as it occurs at run-time. For example, C compilers only perform simple checks to ensure that the syntax is correct, but do no additional checking to determine if the cast is appropriate and will not cause errors.

Unsafe casts include floating-point and integer values to characters (since all characters have an integer ASCII value), numerical arrays to character arrays and casts between pointers and integers. Type-casting allows converting any pointer into any other pointer type, independent of the datatypes they point to. These powerful features makes it easy to control low-level machine details at the cost of sacrificing type safety.

2.6 Memory Leaks

Certain programs fail to release all the memory that they allocate resulting in unnecessary memory consumption over time. This failure to deallocate needless blocks of memory is called a memory leak. These programs will experience a degradation in performance and will eventually crash when they run out of memory. Typical memory leaks involve unreachable dynamically allocated memory as a consequence of having the pointer that pointed to that piece of memory being destroyed.

Attackers can deliberately induce a memory leak to launch a denial of service attack or take advantage of other unpredictable program behavior due to a low memory condition [96].

2.7 Race Conditions

Race conditions are undesired situations that occur as a result of incorrectly moderated accesses to a shared resource.

File-based race conditions such as the time-of-check-time-of-use (TOCTOU) race condition are well known security flaws. Issues crop up when a process checks some property on a file (such as whether it exists or not), then later uses the file with the assumption that the recently checked information is still true. Even if the use comes immediately after the check, there is often some considerable chance that a second process can invalidate the check in a malicious way. This situation can be exploited to launch a *privilege escalation* attack.

For example, a privileged program might open a temporary file "`tmp/foo`" after checking to see that it does not already exist. After the check, but before the file is actually opened, a malicious attacker could replace that file with a symbolic link to the system password file "`/etc/passwd`". The attacker then types his new password file and saves it [91]. Here the attacker has managed to deceive the program into performing an operation that would otherwise be prohibited and has thereby gained elevated privileges.

3 Safe Programming Languages

Safe programming languages are languages in which most of the above vulnerabilities have been made hard or eliminated. By coding in these safer languages it is unlikely that programs will suffer from common security vulnerabilities such as buffer overflows, dangling pointers and format string attacks. To benefit from these languages, programmers need to either implement a program using them or port an existing program into them.

3.1 CCured

CCured is a type-safe implementation of C that statically attempts to verify that source code is free from memory errors, and introduces run-time checks where static analysis does not guarantee safety [62]. CCured seeks to transform C programs into equivalent memory-safe versions, and its main aim is to bring safety to legacy applications [12]. It can also function as a debugging tool as it necessitates that a program be memory safe. The CCured System is shown in Figure 3.

It consists of several components: an OCaml translator, a set of Perl scripts that are used to invoke the CCured application, and a run-time library.

CCured uses a type system that broadens the existing C type system, by differentiating pointer types according to the way they are used in a program. It employs a type-inference algorithm that analyzes the program and is able to deduce the apt pointer type for all the pointers in the program. The intent of this distinction is to prevent misuse of pointers, and thus ensure that programs do not access memory areas they should not. It uses three types of pointers that differ in their speed and capabilities.

Pointers in C programs that have no casts or arithmetic operations performed on them are marked as `SAFE` pointers. Such pointers can be either `NULL` or valid references, and so the only checking that needs to be done are `NULL` checks. Pointers that are not involved in casts but have arithmetic operations performed on them are marked as `SEQ` pointers. `SEQ` pointers carry additional information, such as array bounds details, which are necessary for performing run-time checks. When used, these pointers have `NULL` and run-time bounds checks performed on them. Wild (`WILD`) pointers are pointers whose type cannot be determined statically as they are involved in type-casts, they require `NULL`, bounds and run-time type checking.

This mode of pointer treatment that requires few changes (unlike other safe languages, such as Cyclone) to legacy C code, to make it able to be compiled with CCured is a great advantage of CCured.

CCured prevents dangling pointer dereferences, by using a garbage collector for memory management. Memory is not allowed to be explicitly deallocated (by making `free()` do nothing), instead memory is reclaimed using the conservative Boehm-Demers-Weiser [6] garbage collector. When an object is freed under CCured, the storage is not immediately reclaimed, but rather marked as inaccessible. Subsequent accesses check the mark and signal an error when the object is dereferenced. Ultimately, the mark is reclaimed with the garbage collector to avoid leaks. The garbage collector results in programmers having less control over memory management.

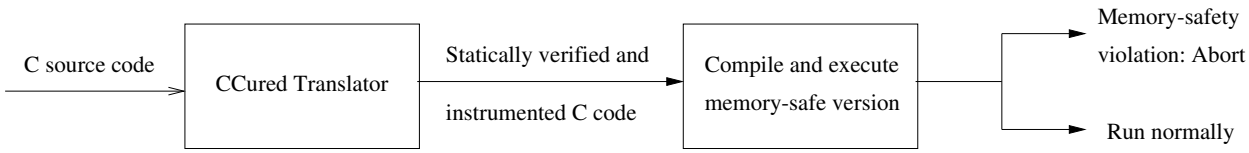

Figure 3: The CCured System

For *type-safety*, CCured's type system is extended *"with a physical subtyping mechanism for handling the upcasts and with a special kind of pointer that carries run-time type information for handling the downcasts"* [15].

CCured achieves compatibility with code that has not been compiled with it by representing arrays and pointers in a compatible format. CCured separates the additional information (referred to as *metadata*) that it maintains for its objects and stores that information in a similar but separate data structure. Each value in the original non-CCured program will be represented by two values in the transformed program – one for data and one for metadata. Similarly, each operation in the original non-CCured program is split into two – one operation on the data value and one on the metadata value. However, integrating some third-party libraries (especially those containing pointers in the data structures) with the CCured type system might be difficult.

The CCured authors carried out experiments to measure the performance cost of run-time checks inserted by CCured and report that, *"for almost all the benchmarks, CCured's safety checks added between 3% and 87% to the running times of these tests."*

Many of CCured's design decisions are due to the fact that it is most concerned with porting legacy code with minimal changes.

CCured works on Linux and Microsoft Windows (Win95 operation is undependable but Win98, Win2k or WinXP should work). It may also work on other systems that use GCC, however the CCured authors have not attempted it. In addition, since the translator is written in OCaml it would require OCaml to be installed in order to run.

3.2 Cyclone

Cyclone [21] is a type-safe dialect of C, that imposes some restrictions to preserve safety, and adds some features to regain common programming idioms in a safe way.

Cyclone retains C's syntax and semantics while preventing most common security vulnerabilities that are present in C programs [46]. It prevents safety violations in programs in a fashion similar to that of CCured, i.e. by using a combination of static analysis for memory management and run-time checks for bounds violations.

In some cases the Cyclone compiler might decline from compiling a program. This could be because the program has been found to be unsafe or because a static analysis of the instrumented source code was not adequate to provide safety guarantees. In such cases, the programmer can modify the program to include more informative annotations that either aid in static analysis, or cause the program to maintain additional information needed for run-time checks.

Cyclone ensures type-safety while endeavoring to maintain low-level control over aspects such as data representation and memory management. Porting legacy C to Cyclone has been found to require an alteration in about 8% of the code.

According to Grossman et al. [33], one of the most interesting aspects of Cyclone is the implemented prevention mechanisms to handle dangling pointer dereferences and memory leaks.

To prevent safety violations, such as `NULL` dereferences, Cyclone introduces new kinds of pointers, such as the "`never-NULL`" pointer, denoted with a "@". Cyclone assures that these pointers will never contain a `NULL` value, and thus `NULL` checks that create an overhead and lead to inefficient programs can be avoided. If a function is called with a pointer that could potentially have a `NULL` value, Cyclone will protect against a possible error by introducing a `NULL` check at the point of function invocation. Functions can be declared to return @-pointers.

Buffer overflows are prevented with restrained pointer arithmetic – pointer arithmetic is not allowed on *-pointers or @-pointers. Alternatively, pointer arithmetic is only supported on new "fat" pointers, indicated by "?". These pointers are called "fat" as their representation consumes more space than *-pointers or @-pointers. For example, instead of writing ``int *p'' in C, we write ``int ?p'' in Cyclone to define a fat pointer p. These pointers are modeled with an address and bounds information, which helps Cyclone determine the size of the array being dealt with, and to appropriately insert bounds checks to ensure safety. Cyclone automatically converts arrays and strings to ?-pointers as needed. Programmers can also explicitly cast a ?-pointer to a *-pointer or to a @-pointer. The former cast causes a bounds check to be inserted and the latter causes a `NULL` and bounds check to be inserted. *-pointers and @-pointers can be cast to ?-pointers in the absence of any checks, converting them into ?-pointers

of size 1. The inserted bounds checks and increased space usage cause fat pointers to be program overheads.

To prevent the use of *uninitialized pointers*, Cyclone performs a static analysis of the source code, and an error is reported by the compiler if it detects that a pointer might be uninitialized. At times the analysis might not be prudent enough to deduce if something is correctly initialized or not. This may result in having the programmer be forced to initialize variables earlier than in C. Uninitialized non-pointers are not treated as errors.

Dangling pointers dereferences are prevented using region-based memory management. All memory in Cyclone is considered to be part of some region; which is defined to be a segment of memory that is deallocated all at once. There are three types of regions – heap, stack and dynamic regions. For each pointer, Cyclone's static region analysis keeps tabs on which region it points into. It also maintains a list of regions that are *live* at any stage in the program. Dereferencing a pointer in a region marked as *non-live* is reported as an error.

Cyclone prevents format string [46] vulnerabilities via the induction of *tagged unions*. Tagged unions are data structures that can take on different types at different times (which is indicated by an explicit tag value). The tags can be used by a function to correctly determine the type of its argument.

Programmers however would need to explicitly add tags to arguments when they call a function, which can become cumbersome. Cyclone provides a feature called "*automatic tag injection*" which guarantees that at compile-time the compiler will add appropriate tag for all arguments.

The performance of Cyclone is fairly good. In some rare cases, Cyclone programs can demonstrate better performance than C programs because of the efficient region-based memory management.

Cyclone version 1.0 is distributed as a compressed archive. It currently runs only on 32-bit machines. It has been tested on Linux, Windows 98/NT/2K/XP using the Cygwin environment, and on Mac OS X. Other platforms may or may not work. To install and use Cyclone, the GNU utilities, including GCC (variant of gcc version 3; version 4 will not work) and GNU-Make are required.

3.3 Vault

Vault [90], like Cyclone, is a safe version of the C programming language with a module system and a novel feature for specifying and checking program resource (represented by keys) use. Programmers are able to control data layout and lifetime of program's resources (such as memory) while being provided with safety guarantees.

An interface is maintained for each Vault module that specifies the names and types of functions and data that are exported. The *rules* for using these functions and data are also maintained by the interface and are checked at *compile time*. Interface rules are recorded using two keywords:

- *tracked* – to identify resources manipulated by the interface
- *change specs* – that show what each function does to those resources.

Violations of these rules will cause the compiler to report an error.

Vault uses *keys* to identify resources, and tracks the state of these resources by associating a *state* with each program point within a function. The state describes the set of keys held and their individual properties at a particular program point.

Conditional access to objects is also supported by Vault. Varying conditions might need to be satisfied at different program points for the object to be accessed, these conditions are described by using keys.

The normal type system is extended to include a *type guard* for different types, which determines if access to an object is permitted. For example, it is possible to specify the availability of operations that can be performed on a variable (say, `f1`) of a specific type (say, `file`) through the usage of a "guard" variable (say, `K`):

```
K:file f1;
```

To access a guarded object, its key must be present in the held-key set (a set of global keys, representing currently available resources). Function types also have conditions indicating which keys must be held prior to the function call and which keys must be set on function return.

Vault's *region-based memory management* system is implemented by using these keys: when a region is created, a key is associated with it and all objects in that region are type guarded by that key. When the region is deleted afterwards, the key is withdrawn from the held key-set and objects in the region become inaccessible.

Potentially dangerous uninitialized variables are prevented in Vault by having every variable initialized either to a explicit programmer specified value or to a default value. Most basic Vault datatypes are associated with a default value. It is illegal for regular "*" pointers to contain `NULL` values. Potentially `NULL` pointers can be declared by suffixing the variable type with a "?".

Type-safety is provided by disallowing arbitrary type-casts. Type-casts are only allowed between values of the following datatypes: `byte, char, short, int, long, long long` and `string` [100].

Vault is a research prototype programming language and not a Microsoft "product". It has limited support and is not

encouraged to be used for critical development projects. The Microsoft C compiler and linker are needed in order to compile Vault programs.

3.4 OCaml

Objective Caml (OCaml) belongs to the Meta-Language (ML) family of programming languages. It is the most popular and extensively used version of Caml – a general-purpose programming language, designed for program safety and dependability [63].

OCaml's set of tools includes an interactive top-level, a bytecode compiler, and a native code compiler.

The "top-level" is an interactive OCaml session that works by reading in expressions, evaluating them and printing out their result. It can be invoked by running the `ocaml` program and is useful for experimentation and quick development. Bytecode compilers allow the creation of portable stand-alone applications out of OCaml programs. The native code compiler ensures good performance and portability through native code generation for major architectures (including IA32, AMD64, PowerPC, SPARC, MIPS, etc.).

Features such as a large standard library, object-oriented programming constructs and modularity make OCaml suitable for large software engineering projects.

Functions in OCaml are first-class citizens, they are treated just like data which allows them to be stored as values in data structures, passed to other functions and returned as the results of expressions (including the return-values of functions).

OCaml is a strongly typed language with a *static type system* which means that the type of every variable and expression in a program is determined at compile-time [39]. This helps to eliminate a significant category of run-time errors that result from type mismatches and also avoids the need for inserting performance hindering run-time checks.

OCaml's *type-inferring* compiler minimizes the need for manual type annotation. The types of functions and variables need not be explicitly declared as they are in C or Java. The compiler is able to infer most of the necessary type information automatically. Type inference eliminates a family of errors which could result in `NullPointerExceptions`, `ClassCastExceptions` and segmentation faults [64]. OCaml however does not perform any implicit type casting. In expressions with mixed datatypes explicit casts are necessary for the expression to be evaluated. OCaml needs this explicit casting to be able to do type-inference correctly and to avoid hard to detect bugs caused by implicit casts.

OCaml has no support for operator overloading, e.g. integer addition is performed using "`+`" whereas floating-point addition is performed using "`+.`". This again is mainly to do type-inference unambiguously. Type inference relies on the concept that each operator has a unique type to determine the types of its operands. If the same operator, say "`+`", is overloaded on integers and on floats, there could be two possible types for the result. It would either be an integer (resulting from integer addition) or a float (resulting from floating-point addition). If the compiler decides on one of these, say integer, later uses of the result as a float would result in an error.

OCaml supports *polymorphic* functions that enhance code re-usability by making it possible to write generic programs that work for values of any type. It is trivial to define data structures that can take on any type of element. A generic function could be written that could be applied to lists of integers or lists of records.

It is a *strictly*[1] evaluated language which means that the arguments to a function are always evaluated completely before the function is applied.

It also has an incremental [2] *garbage collector* so that memory need not be explicitly allocated and freed. There are no `new(), malloc(), delete()`, or `free()` functions. The garbage collector works with two heaps – a minor heap and a major heap. The minor heap, which is garbaged-collected often, holds small objects and objects that are allocated and deallocated frequently. The major heap, which is garbaged-collected infrequently, holds large objects and objects with a long lifetime that are promoted to it after some time from the minor heap.

OCaml programs can be debugged in a variety of ways:

- the *interactive system* can be used to test (small) functions efficiently: various inputs are fed into the interactive system and results are checked for correctness.
- the *function call tracing* mechanism of the interactive system can be used to follow the computation for more complex cases.
- the *symbolic replay debugger* is a debugging tool that allows the program to be paused at any time so that the value of variables and stack layout can be checked.

A foreign function interface (FFI) allows OCaml code to call routines or make use of functions provided by C code. The C code can be statically or dynamically linked with Caml code. It is also possible for C functions to call OCaml functions. The `ocamlmklib` command allows building libraries containing both Caml code and C code.

OCaml version 3.09.2 is available for download. It works on Linux, MacOS X and Microsoft Windows.

1. Strict evaluation is sometimes called "eager" evaluation.
2. Runs in parallel with the application, to avoid detectable delays.

3.5 Haskell

Haskell [34] is a purely-functional general purpose programming language. It is well suited for a diverse set of applications. It allows designing of initial prototypes of programs by writing specifications which can be tested and debugged by actually executing them.

Like most functional languages, Haskell programs are maintainable, as the code is more concise and understandable as compared to that of imperative languages like Java and C.

It is a strongly typed language, eliminating a vast class of compile-time errors. Its polymorphic type system helps enhance code re-usability.

Haskell is a non-strict language with *lazy evaluation*. Lazy evaluation is a technique where only expressions whose results are needed are computed; other possibly unnecessary computations might be delayed or never carried out at all. In particular Haskell uses *call-by-need*: an evaluation strategy, where, if the function argument is evaluated, the results are stored for subsequent uses rather than recomputing them. This has a positive effect on performance.

Haskell has powerful *abstraction mechanisms*, such as the ability to use functions as values, i.e. higher-order functions. Functions in Haskell are first-class citizens. Prudent use of higher-order functions can help build modular programs.

Haskell also features *inbuilt memory management*, which alleviates the programmer from manual memory management issues. Memory is allocated and initialized automatically, and consequently recovered by the garbage collection system. Automatic garbage collection helps prevents run-time errors like dangling pointer dereferences.

The Haskell 98 Foreign Function Interface (FFI) [35] adds support for invoking code written in other programming languages from Haskell and vice versa.

There are a lot of tools for interfacing Haskell with other languages, such as:

- Green Card [32] – a FFI preprocessor for Haskell, simplifying the task of interfacing Haskell programs to external libraries (which are normally exposed via C interfaces).
- HaskellDirect [37] – an Interface Definition Language (IDL) compiler for Haskell, which helps interfacing Haskell code to libraries or components written in other languages (C).
- C→Haskell [11] – A lightweight tool for implementing access to C libraries from Haskell.
- HSFFIG [36] – Haskell FFI Binding Modules Generator, a tool that takes a C library include file (.h) and generates Haskell FFI import declarations for items (functions, structures, etc.) that the header defines.

The drawbacks of using Haskell include the fact that Haskell programs tend to allocate quite a bit of extra memory in the background. In applications where performance and low-level control are desired, an imperative language like C would be a better alternative. Also, functional programming requires an alteration in programmer perspective, which could be difficult.

There are several Haskell implementations, and are distributed under open source licenses. There are currently no commercial Haskell implementations. Haskell compilers and interpreters are freely available for just about any computer.

3.6 Java

The Java programming language *platform* provides a portable, architecture neutral, object-oriented programming language and supporting run-time environment [44]. Java *technology* enables the development of secure, high performance, robust applications on multiple platforms.

Since Java technology was intended to operate in *distributed* environments, security features have been designed and built into the language and run-time system. Java applications are resistant to malicious code injections [3]. The language helps programmers write safe code by featuring type-safety, automatic memory management, garbage collection, and bounds checking on strings and arrays [45]. There are no explicit programmer-defined pointer datatypes and no pointer arithmetic.

The Java security model is based on a customizable *sandbox* which is a restricted environment in which Java programs can run untrusted remote code, without potential risk to systems or users [82].

For portability, the compiler generates *bytecodes* – an architecture neutral intermediate format. The same bytecodes will run on any platform and there are no datatype incompatibilities across architectures.

Java provides extensive compile-time checking, followed by a second level of run-time checking. Java is strict in its definition of the basic language, it specifies the sizes of its basic datatypes and the behavior of its arithmetic operators.

The *Java Virtual Machine (JVM)* is the specification of an abstract machine for which Java compilers can generate code. Implementations of the JVM for different platforms provides the concrete realization of the virtual machine.

The Java garbage collector runs as a low-priority background thread, ensuring a high probability that memory is available when needed. Calculation-intensive sections of large programs can be written in native machine code to help

3. It has however been demonstrated that a single-bit error induced in a Java program's data space can be exploited to execute arbitrary code [31].

improve performance. Although, the portability and security features of Java are lost via this.

Bytecode verification ensures that code conforms to the JVM specification and guarantees that only valid bytecodes are executed, preventing hostile code from corrupting the run-time environment. Run-time safety is guaranteed by the bytecode verifier in conjunction with the JVM.

Secure Class Loading provides security by associating classes loaded by a particular class loader with a unique namespace. A *namespace* is a set of unique names of the classes loaded by a particular class loader. The namespace of classes loaded by a particular class loader is isolated from the namespaces of other class loaders. As trusted local classes and untrusted classes downloaded from remote sites are loaded through separate class loaders, the possibility that an untrusted class can substitute a trusted class and thereby launch an attack is reduced.

The JVM arbitrates access to critical system resources and uses a *SecurityManager* class to minimize the actions of untrusted pieces of code.

Java also has comprehensive `APIs` and built-in implementations of important security standards, which help in building secure applications. There is support for a wide range of cryptographic services, development and deployment of public key infrastructure, secure communication, authentication, and access control [82].

Java programs may however have lower performance than their counterparts in languages such as C/C++. This is because bytecode is not as optimized as the machine code generated by C/C++; and the use of a automatic garbage collector that has to decide when to delete objects can be more expensive than imperative deletion in C/C++.

The Java Software Development Kit (JDK) is required for building applications, applets, and components using the Java programming language. It includes the Java Runtime Environment (JRE) which consists of a Java virtual machine, class libraries, and other files that support the execution of programs written in the Java programming language.

3.7 .NET Framework Architecture

The .NET Framework [59] is a software development platform, similar to Java, created by Microsoft. .NET technology allows development in multiple programming languages along with an extensive standard library.

The .NET Framework uses an intermediate language known as the Common Intermediate Language (CIL) to be platform independent. Programming languages on the .NET Framework first compile into CIL and are thereafter compiled into native code using just-in-time (JIT) compilation. The amalgamation of these concepts is a specification called the Common Language Infrastructure (CLI), which has components for: exception handling, garbage collection and security. Common Language Runtime (CLR) is Microsoft's implementation of the CLI.

.NET's security mechanism has two features: code access security, and validation and verification. Code access security uses the source[4] of the assembly[5] to determine the permissions to be granted to it.

The CLR performs the validation and verification tests on the loaded assembly. Validation ensures that the assembly contains valid metadata and CIL. The verification mechanism checks to see if the code attempts to do anything that is "unsafe". Unsafe code will generally only be executed if the assembly is installed on the local machine.

The .NET framework is primarily supported only by the Microsoft family of products, including the Windows Server System, the Windows XP operating system, and the Microsoft Office System. However, there are several open source development projects, such as Mono [61] (a project led by Novell, Inc.) that provide software for supporting .NET on different operating system platforms including Linux, Solaris, Mac OS X, and BSD (OpenBSD, FreeBSD, NetBSD).

The .NET Framework SDK is free and includes compilers and various other utilities to aid development. Several languages have compilers for the .NET Framework, but only a limited set is predominantly used and supported. Prime among these are C#, Visual Basic .NET, C++/CLI, J#, JScript .NET. The Visual Studio .NET [60] integrated development environment is the major tool used for development. The Express Edition is available as a free download, but the Standard, Professional and Team editions are available for a fee.

3.8 Scheme

Scheme [78] is a statically scoped dialect of the Lisp programming language. It is a small but powerful and productive language, designed to have clear and simple semantics. It supports a wide variety of programming paradigms, including imperative, functional, and object-oriented.

Like most higher level languages, Scheme is memory-safe, in the sense that one cannot leak memory, nor write off the end of a buffer.

Most Scheme systems are interactive, allowing programmers to incrementally develop and test parts of their program. It can also be compiled, to make programs run fast. Scheme is

4. Source refers to whether it is installed on a local machine or downloaded from a remote site.

5. An assembly is the building block of a .NET Framework application; they are complied and form an atomic functional unit that can be deployed. Assemblies exist as executable program files (.exe) or dynamic link library (.dll) files.

not usually quite as fast as C, and results can vary depending on the compiler used.

However there are several significant limitations that make it unsuitable for writing large-scale security-critical code. Scheme is a strong, dynamically typed language. Every value has a type which can be ascertained; however it does not provide static checking of the types, instead all type checking is done dynamically, at run-time, which means that there are no assurances that the program will not apply an operation at run-time to arguments for which it makes no sense. This lack of an expressive type system makes it harder to safely compile Scheme to an efficient native form.

There are no standard exception handling semantics; the standard stipulates that certain actions result in an error, but there is no standard facility for handling such errors when they occur.

Scheme programs can make use of new derived expression types, called macros. Scheme provides a *hygienic* macro system which is safer and often easier to work with than the fully programmatic Lisp-style macros, though not as powerful. Hygienic macros [50] are macros whose expansion is guaranteed to not cause name clashes with definitions already existing in the surrounding environment. However, if macros are not used judiciously the syntax of the program can change in undesired ways making it difficult to understand and semantically analyze what the program does.

Since the Scheme standard is very conservative and specifies only the core language, many different implementations have been developed. The Scheme community is still considerably fragmented and so there is a very limited standard library. Most libraries work only in specific implementations. The Scheme Request For Implementation (SRFI) [79] project aims at resolving this.

Most Schemes have FFI support to the native language (i.e. the language used to implement the Scheme interpreter/compiler) allowing native code to be called from Scheme and vice versa [80].

There are several implementations of Scheme, both free and commercial, that run on various hardware platforms and operating systems.

4 Overcoming Security Flaws in C

Despite all the security vulnerabilities known to be present in the C programming language, it is still widely used for software development and has proved to be efficient and indispensable to programmers. Developers continue to use C because of the trade-offs between safety and functionality. Potentially unsafe code is preferred over code that is inefficient, more memory-greedy and has control to a lesser extent over low-level data structures and memory management.

Features of the C language such as pointer arithmetic, typecasting of pointers and other memory operations that can directly access raw memory make it suitable for writing low-level system programs which need high performance [66].

Other factors in favor of C include - programmer experience, familiarity, portability and extensibility (it is easier to use existing C libraries).

However, as application requirements have become increasingly sophisticated, C programs have had to contend with frequent convoluted pointer computations, which leads to efficient but insecure programs, and it is common belief that safety violations are likely to remain prevalent in C programs.

The infamous buffer overflows, described as the "Vulnerability of the Decade" [20], are so widespread in C because it is inherently unsafe. There is no automatic bounds-checking for array and pointer references, and so, the onus is on the programmers to perform these checks. Also, many of the standard C library functions (such as, Strcpy(), sprintf(), gets()) are themselves unsafe and hence the programmer is responsible for bounds-checking these too. The process of inserting checks that cover all possible paths is tedious and error-prone, and hence programmers often omit many of these checks. There could also be situations where programmers include checks for all their code but use a dynamically linked library which is linked in with the executable at runtime and so are provided with no assurances about the safety of the C code within the library.

In this section we present a survey of the diverse countermeasures that either seek to eliminate particular vulnerabilities or prevent them from being exploited while maintaining the functionality of the system.

The countermeasures are divided into several categories based on how they tackle the problems.

4.1 Static Analysis of Source Code

Static source code analyzers are automated tools that attempt to find software vulnerabilities by examining program source code without having to execute the program.

They are primarily designed to be used at development time to diagnose common coding errors that contribute in making code unsafe. They can be used as a first step in manually auditing source code. Using these tools is relatively easy; they take as input one or more source code files, and after inspecting them, produce an output which identifies potentially vulnerable areas of code.

Detecting certain vulnerabilities, such as buffer overflows, by statically analyzing source code is, in general, an undecidable problem. The Halting Problem can be reduced to the

buffer overflow detection problem [52]. Thus, analyzers will generate a lot of false positives (report bugs that the program does not contain) and/or false negatives (does not report bugs that the program does contain). However, they are still capable of producing useful results. The major advantage of static analyzers is that security flaws can be eliminated before the code is deployed.

As proposed by Younan et al. [100], static analyzers can be fractioned into two classes based on whether they need the source code to be semantically commented (*annotated*) or not.

4.1.1 Analysis of semantically commented code

This class of static analyzers exploit semantic comments in source code and libraries to determine assumptions and intents being made in the code. Semantic comments follow a definitive syntax and provide extra information that is useful in determining programmer intentions. For example, programmers would annotate a pointer with `/*@notnull@*/` to indicate that the pointer's value cannot be `NULL` at that program point. The convenience of this approach is that it allows efficient scanning of the source code for assumption violations.

Splint

Secure Programming Lint [86] (successor to LCLint) is a tool for statically analyzing C source code for security vulnerabilities and coding flaws. It can efficiently detect a broad range of implementation errors by exploiting annotations that describe assumptions made about a type, variable or function interface.

Splint's checks include type mismatches, unreachable code, buffer overflows, `NULL` and dangling pointer dereferences etc. Better checking results can be obtained by putting more effort into annotating programs [87].

Known vulnerable functions in the standard library are annotated with conditions that must be met before and after the function is called.

For example, `strcpy()` is annotated as follows:

```
/*@requires maxSet(s1) >= maxRead(s2) @*/
/*@ensures maxRead(s1) == maxRead(s2) @*/
```

The `requires` clause signals that the buffer `s1` must be large enough to hold string `s2`. The `ensures` clause signals that `maxRead` of `s1` after the call is equal to `maxRead` of `s2`.

Splint allows suppression of error messages in code regions commented with `/*@ignore@*/` and `/*@end@*/`.

Format string vulnerabilities are detected using `taintedness` attributes. These attributes record whether or not a `char *` came from a possibly untrustworthy source and are useful in preventing several security vulnerabilities. All user input is considered tainted and a run-time error reported before a tainted value is used in an unsafe way [26].

Splint is used on source code in an iterative manner. After the first run, the code or annotations are suitably modified based on the warnings produced. In subsequent runs, the modifications are checked and newly documented assumptions disseminated. This process is continued until Splint issues no warnings.

Splint is compiler independent and is available as source code and binary executables for several platforms (including Linux x86, FreeBSD, OS/2, Solaris, Win32) and can be freely distributed and modified under the GNU General Public License.

4.1.2 Analysis of code lacking semantic comments

These analyzers attempt to verify if code is safe by constructing an execution model at every program point or by condensing the program to a constraint system. By tracing the different execution flows and actions, they are able to demonstrate if particular conjectures always withstand.

ITS4

It's the Software Stupid! Security Scanner [42] is a command-line tool for statically scanning C and C++ source code for possible security susceptibilities. ITS4 scrutinizes source code for function calls that are likely to be perilous, such as `strcpy`. It does *lexical analysis* on the source code by breaking it up into a stream of tokens and then examining the resultant token stream against a vulnerability database that contains a list of unsafe and misused functions [91].

At startup, the vulnerability database is read from a text file – the contents of which are stored in memory while the tool is in use. Vulnerabilities can be added, deleted and modified from the database.

ITS4 also guards against `TOCTOU` file-based race conditions.

For each case, ITS4 generates a report that includes a brief description of the problem and suggestions on how to overcome it. Possibly vulnerable functions that are identified are further analyzed to determine the severity level they should be reported with.

For example, `strcpy(buf, "\n")` will be reported with the lowest severity as the second argument is a fixed string which does not pose much of a threat. ITS4 works adequately fast to provide real-time feedback to programmers while coding.

ITS4 can be customized for specific applications using its configurable command-line options which are useful for focusing on specific functions while suppressing others.

It is freely available for non-commercial use on Unix and Windows platforms.

RATS

Rough Auditing Tool for Security [73] is an open source tool that supports five popular programming languages: C, C++, Perl, PHP and Python. It scans source code looking for known insecure function calls and can detect errors such as buffer overflows and race conditions in file accesses, such as the `TOCTOU` problem.

As can be inferred from the name, RATS performs only a *rough* analysis of source code. It does not find all errors and may also report false positives.

RATS scans each file specified on the command line, and produces a report (text, HTML or XML) when scanning is complete. Vulnerabilities reported depend on the data contained in the vulnerability database(s) used, and the warning level in effect. Filenames and line numbers of vulnerability occurrence are reported along with a short description of the vulnerability, severity and recommended actions. Simple analysis to eliminate reporting conditions that are evidently not problems are also performed.

RATS is free software and is released under the GNU Public License (GPL). It was developed for Unix and has been ported to Windows as well.

BOON

Buffer Overrun detectiON [7] is a static analyzer for detecting buffer overflows in C code. It treats strings as abstract datatypes (so as to recognize natural abstraction boundaries that are obscured by the C string library) and models strings buffers using a pair of integers: the number of bytes allocated for the string (its *allocated size*), and the number of bytes currently in use (its *length*) [94]. The buffer overflow problem is in effect reduced to an integer range analysis problem.

The integer ranges are used to define a *constraint language*, and all string operations are modeled based on what they do to the *allocated size* (`alloc(s)`) and *length* (`len(s)`) of the string. The program is then parsed, and for every statement a set of integer range constraints is developed. The safety property to be checked is:

`for all strings s, len(s)` $\leq$ `alloc(s)`

Next, the constraint system is solved and matched to detect inconsistencies, which are reported as possible overflows. The programmer then needs to manually check the source code and see whether they are real overflows or not.

The tool generates a considerable number of false positives due to a trade off of precision for scalability. There are also some restrictions when analyzing pointer operations and no support for format string vulnerability detection.

BOON is available under the BSD license for Unix platforms. The current release of BOON is provided on an "as is" basis and is only a research prototype that is known to have limitations and probable bugs. It is unsupported and its use may require skillfulness and diligence.

Prevent

Coverity's Prevent [70] is a sophisticated static analysis tool, that helps make new and legacy source code more stable and secure. It employs a dataflow analysis engine to detect flaws in C and C++ code. It spawned from Stanford University's xgcc/Metal research [58].

Prevent automates the process of scanning, reporting and tracking for security vulnerabilities, and covers various categories of software defects, such as [97]:

- Run-time crash causing defects: including `NULL` pointer references, use after free and double frees.
- Poor performance: including memory leaks, file handle leaks, database connection leaks and misuse of API's.
- Incorrect program behavior: including uninitialized variables and invalid use of negative values.
- Security vulnerabilities: including buffer overflows, integer overflows and format string vulnerabilities.

The analysis engine models at compile-time, the outcomes that the operations in the source code could have at run-time. Simulation is done for each program path using a finite state machine. The analysis engine has a core set of software checks incorporated into it, but can also be customized for specific security requirements.

The analysis is carried out by searching individual lines of code in isolation for syntactic anomalies, as well as across complex code processes and functions. This helps in presenting a complete view of each event that led to a vulnerability. It has a low false positive rate of about 20%.

It provides summarized details of potential risks, via email or web graphical user interface, including defect volume, location, dispersion and severity. It also has a "Defect Manager" with "CodeBrowser" that can be used to navigate the code to view and fix the bugs.

Prevent automatically integrates into a variety of build environments and requires no changes to the code or the build processes. It supports several platforms (including Linux, HPUX, FreeBSD, NetBSD, Windows, Mac OS X, Solaris Sparc, Solaris x86) and several compilers (including GCC, G++, Sun CC, MS Visual Studio, Intel Compiler for C/C++). It currently works with various open source projects, including FreeBSD, MySQL and Mozilla.

Prevent is an enterprise tool, and its price is based on the total code size to be analyzed. However, it offers a free code

audit to certify and prove its capabilities.

Fortify Source Code Analysis Suite

The Fortify Source Code Analysis Suite [30] is an integrated set of static source code analysis tools, similar in many ways to Coverity's Prevent. It works seamlessly with existing development and audit tools and processes and allows for effective scanning, tracking and fixing of software security flaws.

It does an effective and precise data flow analysis of source code, and provides an interactive graphical view of the discovered security issues. It can efficiently processes voluminous and complex code bases.

Software developers can use Fortify's plug-and-play capabilities with popular Integrated Development Environments (IDEs) including Microsoft Visual Studio, Borland JBuilder and Eclipse to dispense security vulnerabilities early in the development lifecycle.

The tools work much like a compiler. Minor modifications to the build script invokes Fortify's language parser which reads in a source code file(s) and transforms them to an intermediate format, optimized for security analysis. This intermediate format is exploited by the Analysis Engine to locate security flaws.

The Analysis Engine, comprises of four distinctive analyzers:

- Data Flow Analyzer – discovers possibly unsafe data paths.
- Semantic Analyzer – detects use of insecure functions or procedures and infers their context of use.
- Control Flow Analyzer – tracks ordering of operations to detect unsuitable coding constructs.
- Configuration Analyzer – tracks vulnerable interactions between configuration and code.

It also provides a "Rules Builder" so users can extend and customize the capability analysis rules.

It supports multiple operating systems (including Linux, Windows, Mac OS X and Solaris) and several programming languages (including C, C++, C#, Java, JSP, PL/SQL, ASP.NET, VB.NET and XML).

The Analysis Suite can be purchased as an Enterprise Edition or a perpetual license, and the prices are per CPU on the build server. It also offers a free code audit to certify and prove its capabilities.

Prexis: Automated Software Security Assurance

Ounce Labs' Prexis suite [67] is a set of automated software tools for static source code security analysis.

Prexis/Engine is the source code analysis and security vulnerability knowledge-base core. It uses advanced complier technology and a customizable software security knowledge-base. It provides a complete characterization of software risks, coding faults, design defects and policy violations.

The specialized compiler technology is used to parse the source code to derive a Common Intermediate Security Language (CISL). The intermediated code is analyzed by Ounce Labs' Contextual Analysis technology to detect, confirm, and categorize the vulnerabilities. The results are then stored in a database for analysis and reporting.

The Contextual Analysis technology allows source code to be automatically analyzed in detailed depth. The context in which a call is used influences whether or not it is unsafe. Vulnerabilities are determined by tracking the flow of data through an application and understanding the interrelationships between the different program elements.

The security knowledge-base, with over 60,000 entries, allows Prexis/Engine to identify a wide range of vulnerabilities including buffer overflows, insecure access control, privilege escalations, race conditions, SQL Injection etc. The security knowledge-base is customizable to specific security and policy criteria.

Prexis uses a metric called V-Density (vulnerability density), a numeric expression computed by associating the number and criticality of vulnerabilities to the size of the application being analyzed. Once the V-Density has been determined, thresholds can be set for understanding the security state of critical software.

Prexis supports multiple operating systems (including Linux, Windows and Solaris) and several programming languages (including C, C++, Java and .NET languages).

It is proprietary software and can be purchased for a fee.

4.2 Dynamic Analysis Tools

Dynamic analyzers instrument source programs and create a version that when run, has the same behavior as the original program but generate specific events when a possible vulnerability is encountered. As the checking is done dynamically, more accurate checking than can be done statically is possible. This is because of the precision of the information that they provide as compared to static analysis tools. Static analysis tools usually report errors that are only approximations of the properties that actually hold when the program runs [43]. Hence, the high false positives and/or false negatives that are generated by static analyzers can be eliminated by using dynamic analyzers. However, some errors might be omitted as some execution paths might never have been followed while analyzing.

These tools are primarily designed to be debugging tools

for finding memory leaks and buffer overflows. Using these tools is simple, programs to be tested are linked against them and they generate a report detailing errors and other significant events. These tools generally have a high performance overhead and so are mainly used for debugging purposes.

Purify

Purify is a tool designed to detect memory bugs, such as leaks and access errors, which are a source of many security vulnerabilities. It performs verification dynamically; i.e. it detects errors at program run-time.

Purify parses and adds verification instructions into compiled object code, including third-party and operating system libraries. This helps the program to output the precise position of the error, the memory address affected, and other related data when a memory error occurs. The instrumented program's object code will have function calls that check every memory read and write for various types of access errors, including uninitialized memory reads and freed memory reads/writes. *"Purify tracks memory usage and identifies individual memory leaks using a novel adaptation of garbage collection techniques"* [38].

The inserted function call instructions maintain a memory state table, in which two bits are used to associate one of three states with each byte of memory in the heap, stack, data and BSS sections. The three states are:

- unallocated (unwritable and unreadable),
- allocated and uninitialized (writable but unreadable)
- allocated and initialized (writable and readable).

A read/write to unallocated bytes causes a warning message to be printed. Writing to memory marked as allocated-and-uninitialized causes it to change state and become allocated-and-initialized. Heap overflows are detected by allocating *"red-zones"* at the start and end of memory blocks returned by `malloc()`. Red-zone bytes are marked as unallocated, and so an access to these bytes will signal an error.

Purify works automatically, i.e programs linked with Purify will be implicitly verified for memory errors. This is in contrast to traditional memory debuggers that essentially need to be done by hand, by stepping through the code line by line.

It is beneficial to use Purify on programs written in programming languages that have manual memory management (such as C) as there is a higher probability of having memory leaks here, as compared to programs that have automatic memory management (such as Java) where memory leaks are implicitly avoided. As it is targeted mainly for debugging purposes, its relatively high overheads are not an issue.

Purify is available for Unix (Solaris, SPARC, Ultra SPARC), Windows (2000, XP Professional, NT 4.0) and Linux (Red Hat, Enterprise) environments, but is proprietary software and can be expensive to license [40].

Valgrind

Valgrind [89] is a set of tools for automatic memory debugging and profiling of large programs. Detection of memory bugs help make programs more stable and profiling helps in efficient memory use of programs.

Valgrind is essentially a virtual machine using just-in-time (JIT) (i.e. dynamic binary translation) compilation. This means that applications do not need to be modified or recompiled to use Valgrind, it can even be used on programs for which only binaries are available and there is no source code. Valgrind initially translates the program into an intermediate, more elementary form called *ucode*, which is instrumented by one of the tools. The ucode is then translated back into x86 code that is run on the target machine.

A significant amount of performance is lost in these transformations and by the code that the tools insert. Programs run considerably slower under Valgrind, the slowdown factor can range from 5–100 depending on the tool used. Since it is intended mainly as a debugging tool, this slowdown is acceptable.

Valgrind works with programs written in any language, though mainly aimed for programs written in C and C++, it has been used on programs written in Java, Perl, Python, assembly code, Fortran, Ada, and many others as well. Using Valgrind is straightforward, the program to be run under Valgrind is prefixed with `valgrind --tool=tool_name` on the usual command-line invocation.

The Valgrind distribution includes five useful debugging and profiling tools: Memcheck, Addrcheck, Cachegrind, Massif, and Helgrind. Valgrind is *extensible*, which means that new tools that add arbitrary instrumentation to programs can be written and plugged in.

Valgrind is free open source software, available under the GNU General Public License. It is not distributed as binaries or RPMs, instead the source code has to be downloaded and compiled in order to be installed on the system. It is actively maintained and supported on x86/Linux, AMD64/Linux and PPC32/Linux platforms.

4.3 Sandbox Security

Sandboxing seeks to restrict the corruption that the exploitation of a vulnerability might result in. It does not thwart the vulnerability or its violation, but instead tries to limit the amount of damage that a compromised component can cause to a system.

Sandboxing employs the "Principle of Least Privilege" [81], according to which, an application is granted the

least possible privileges to be able to complete its job, and the privileges are granted only for the least amount of time necessary. It is usually implemented in two ways:

- *Seclusion of faults:* guarantees that when a program component fails, it will not result in total system failure. Address spaces of different modules are usually kept separate to enforce fault isolation, however for tightly-coupled modules this incurs substantial execution overhead, due to costly context switches and inter-module communication.
- *Imposition of a policy:* a policy is defined and enforced, stating categorically what an application is allowed and prohibited from doing. The enforcement is usually done via a reference monitor where an application's access to specific resources is regulated.

The main drawback of this type of countermeasure is that it requires a well-reasoned, comprehensive policy regarding what can and cannot be accessed. Creation of such a policy usually requires a detailed understanding of the program being sandboxed. A further problem with policy-based sandboxing technologies is that they are not broadly portable [92].

Systrace

`systrace` [88] is a utility that audits and regulates an application's access to a system by generating and enforcing access policies for system calls. It obviates the requirement to run a program in a completely privileged mode. Using `systrace`, programs can be run unprivileged but are provided with facilities for *privilege elevation* when required. A configurable policy determines which operations can be executed with elevated privileges [71].

`systrace` is especially useful when running untrusted binary-only applications, the access of these applications to the system can be sandboxed, increasing the system's total security.

The policy specifies the behavior desired of applications on a *system call* level basis. With `systrace`, a user can decide which programs can make which system calls and how those calls can be made. Policy generation is possible either *automatically* – a base policy is generated, containing all the system calls the application wishes to make, this list can later be refined; or *interactively* – the user decides whether an attempt to execute a system call that is not described by the current policy can be performed during program execution. Operations that are not explicitly permitted by the policy are denied by `systrace`. Such operation denials are logged and the user can decide whether he wants to add it to the presently configured policy or not.

Like most existing utilities and tools `systrace` does not guarantee complete security, but the additional layer of security that it introduces makes it more difficult for an attacker to gain unwarranted access.

`systrace` is distributed under a BSD-style license and ships by default with NetBSD, OpenBSD and OpenDarwin. There are also ports for Mac OS X (currently unmaintained), FreeBSD, and Linux.

SFI

The Software-based Fault Isolation [95] approach implements fault isolation using a single address space. This negates the need for context switches and allows for cheaper inter-module communication, but increases execution time. Only distrusted modules warrant an execution time overhead. Code in trusted domains execute unvaried.

The approach has two parts. First, a distrusted module's code and data is loaded into its own *fault domain* – a logically differentiated section of the application's address space, comprising a contiguous region of memory. Each fault domain has a unique identifier which is used to mandate its access to process resources.

A fault domain is split up into two segments, one for the code of the distrusted module and the other for its static data, heap and stack. All virtual addresses within a segment share a distinct pattern of upper bits, called the `segment identifier`. Second, the distrusted module's object code is instrumented to prevent it from writing or jumping to an address outside its fault domain. Such isolated modules are not capable of modifying *"each other's data or executing each other's code except through an explicit cross-fault-domain RPC interface"* [95].

This isolation is enforced using two techniques:

1. *Segment matching* inserts checking code before every unsafe instruction i.e. an instruction that jumps or writes to an address that cannot be statically verified to be in the right segment. The checking code decides if the target address of the unsafe instruction has the correct segment identifier. If the check is unsuccessful, an error will be reported. This technique allows the programmer to exactly locate the violating instruction.

2. *Address sandboxing* attempts to reduce the overhead associated with enforcing SFI, by providing no information on the source of faults. Every unsafe instruction is preceded with inserted code, that sets the upper bits of the target address to the correct segment identifier. Writable memory is allowed to be shared across fault domains, using a technique called *lazy pointer swizzling*: for each address space segment that requires access, the page tables are modified so as to map the shared memory at the same offset.

These techniques make it more difficult for malicious users to exploit memory address vulnerabilities.

Program shepherding

Program shepherding [47] is a technique that inspects all execution-flow transfers throughout program execution to assure that each respects a given security policy. The focus is on preventing transfer of control to malicious code, thereby preventing a wide range of security attacks. For example, buffer overflow attacks are prevented, as a successful attack would need a control-flow transfer that would violate the security policy.

Program shepherding can also be used to disallow execution of shared library code except through declared entry points, and can ascertain that a return instruction only returns to the instruction after the point of the call.

Program shepherding is implemented via three techniques:

- *Execution privileges restricted based on code origins*: this can ensure that malicious code disguised as data is never executed. Code origins are checked against a security policy to see if it should be granted or denied execute privileges. Code origins are sorted based on whether it is from the original image on disk and unmodified, dynamically generated but unmodified since generation or code that has been modified .
- *Limited transfer control*: this denies attackers the possibility of branching directly to their code and executing it, shunting any sanity checks that might have been required to be performed before that code was executed. Enforcing the execution model involves allowing each branch to jump only to a specified set of targets.
- *Un-Bypassable Sandboxing*: shepherding guarantees that sandboxing checks placed around any type of program operation will never be dodged. This helps in providing complete security, as an attacker that acquires control of the execution, cannot sidestep the checks and skip straightaway to the sandboxed operation. This is possible due to the control transfer restrictions.

The shepherding techniques have been implemented in DynamoRIO (Runtime Introspection and Optimization) [8, 9] a run-time code modification system that allows code transformations on any portion of a program, while it executes. *"The resulting system imposes minimal or no performance overhead, operates on unmodified native binaries, and requires no special hardware or operating system support"* [48].

To reduce interpretation overhead, program shepherding performs security checks only once, and if the code complies it is placed in a *code cache*. The code cache is protected from malicious modification, so future executions of the trusted cached code proceed with no security overhead. This leads to efficient execution.

The program shepherding authors point out that program shepherding could be used to allow services provided by the operating system to be moved to more efficient user-level libraries.

They cite the Exokernel [24] class of operating systems as an example, here program shepherding could enforce unique entry points to the unprivileged libraries that provide the normal operating system operations, thereby giving users efficient control of system resources without sacrificing security.

The DynamoRio binary package can be downloaded from the DynamoRIO Release website [23]. It is beta software and is supported on the following platforms: Linux (RedHat 7.2, RedHat Enterprise Linux WS 3, Fedora Core 2, Fedora Core 3) and Windows (NT, XP, 2003, 2000).

4.4 Compiler Techniques

The compiler plays a crucial role in determining the execution environment of a program. Numerous modifications to overcome security vulnerabilities in programs can be made in the compiler, without necessitating change in the programming language in which the programs are developed [100]. These techniques generally work by performing bounds checking on C programs. The drawback however is that performance critical pointer-intensive programs will be considerably slowed down [56].

To use these tools the compiler usually needs to be patched with them. The protection offered by these tools can then be enabled or disabled via specific flags.

SCC: The Safe C Compiler

SCC [2] is a C source-level program transformation system which includes checks for all pointer and array accesses so as to provide efficient detection of memory access errors in unchecked C code. The technique is to have *safe pointer* representation by using a data structure that contains safety information to model pointers; and by inserting run-time checks.

A *safe pointer* contains the value of the pointer along with supplementary information called *object attributes*. Such attributes include, base address, size of the memory object in bytes, storage class (heap, local or global) and capability: a unique identifier for the storage allocation of dynamically allocated variables.

```
typedef {
   <type> *value;
   <type> *base;
    unsigned size;
    enum Heap=0, Local, Global storageClass;
```

```
  int capability; /*plus FOREVER and NEVER*/
} SafePtr<type>;
```

Memory bounds errors are checked at run-time by comparing against the base and size fields of the extended pointer structures.

The compiler transforms conventional C programs in three steps:

- *pointer conversion:* pointer definitions and declarations are transformed to the extended structures that incorporate object attributes.
- *run-time check insertion:* checks are inserted ahead of every pointer and array access to detect memory access errors.
- *operator conversion:* operations on pointers must be modified to interact properly with the extended safe pointer structure.

The run-time library prevents dangling pointer dereferences by maintaining a *capability store*. The capability store keeps track of all dynamically allocated storage by monitoring allocation, deallocation and memory access events [53]. The set of capabilities in the capability store represents all the dynamic storage that is currently active.

Safe-C requires few modifications to C source code and is a good tool for making legacy code safe. It however provides limited safety guarantees at a high price (130% to 540%) due to exhaustive run-time checks (memory bounds checks and capability database queries). Also, the extended pointer structures increase the spatial overhead, as each pointer now holds additional information. It is mainly useful for program debugging.

The source release for SCC version 1.0.0 is not publicly available, details on its distribution can be obtained from its webpage [77].

Fail-Safe ANSI-C compiler

The Fail-Safe ANSI-C compiler [66] supports the full ANSI C standard while preventing unsafe operations. The authors of the compiler define an "unsafe operation" to be an *"operation that leads to "undefined behavior", such as array boundary overrun and dereference of a pointer in a wrong type."*

The compiler introduces run-time checking code into programs to prevent corruption of memory data structures by buffer overflows or dangling pointer deferences. Violations of these checks causes an error to be reported and program execution to be terminated.

The compiler uses *fat pointers* which are a combination of three values, denoted by `ptr(b, o, f)`, where `b` is the base address of the contiguous region it is pointing to, `o` is the offset to that region and `f` is a *cast flag* that indicates if the pointer *may* refer to a value other than its static type.

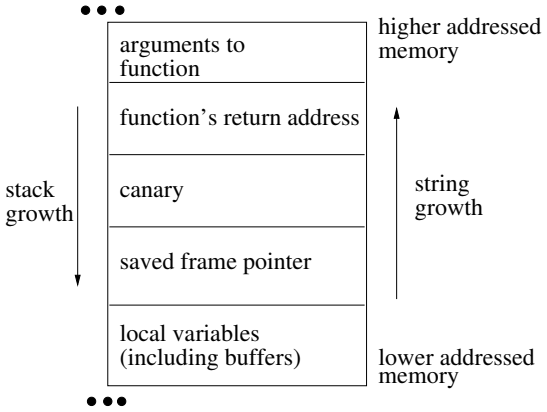

Figure 4: The StackGuard Stack Structure

The integer rendering of a pointer is its base + offset. When a pointer is cast to an integer the result is a *fat integer* of the form – `ptr(b, o, 1)`, allowing the integer to be cast back to a pointer if desired.

Dereferencing of invalid pointers (pointers that refer to a location outside of a valid region) is prevented by insertion of bounds checks. These checks use the base address and offset of the pointer.

If the cast flag of a pointer is unset (i.e. f = 0), the correct type of value is guaranteed to be read and hence no type checking is required during such pointer dereferences. Alternatively a value read using a pointer with f = 1 may have an incorrect type, and hence type checking of such values is required.

Dangling pointer dereferences are prevented by marking freed memory regions as "already released" but not actually releasing them. The size field of such fields is set to zero. This forbids access to that block, preventing dangling pointers.

Programs compiled by the current Fail-Safe C compiler have been found to be 30% to 500% slower than the original C programs [65].

Details about the Fail-Safe C Project can be obtained from its webpage [27].

StackGuard

StackGuard [19] is an extension to *gcc* (GNU C Compiler, part of the GNU Compiler Collection), that can be configured to either detect or prevent stack smashing attacks.

To modify the function's return address, buffer overflow attacks need to overwrite all the stack data contained between the overflown buffer and the higher addressed function return address. Stackguard attempts to detect a return address modi-

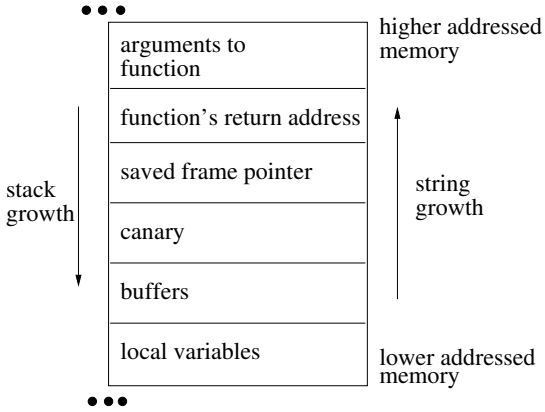

Figure 5: The ProPolice Stack Structure

fication by placing a "canary" (known value placed between a buffer and control data) before the return address on the stack. When the function returns, the canary value is checked prior to jumping to the return address, if it has been altered an error is reported and the program terminated, if it has not, the canary is removed and the function returns normally. The StackGuard stack structure is shown in in Figure 4.

Stackguard supports three types of canaries:

- Random canary – a random value (32-bit number) unknown to the attacker that is generated at program initialization.
- Terminator canary – a known value built of termination symbols for standard C library functions; 0 (`NULL`), CR, LF, and -1 (`EOF`).
- Random XOR canaries – random canaries that are XORed with the return address. If either value is corrupted the canary value is wrong.

There are several ways to bypass StackGuard [74].

StackGuard existed as patches for gcc versions up to 2.95, but is no longer available.

ProPolice

The "Stack-Smashing Protector" or SSP, also known as ProPolice [25] is a patchset for *gcc*, designed to protect applications from buffer overflow attacks, based on a protection method that automatically inserts checking code into an application at compile time.

The main ideas are the rearranging of local variables so as to place pointers before buffers in memory so as to avert pointer corruption, and the skipping of instrumentation code from some functions to reduce performance overhead.

ProPolice places the "canary" before the frame pointer so as to detect attacks that leave the return address intact but modify the frame pointer [49]. It supports only a random XOR canary. The ProPolice stack structure is shown in in Figure 5.

Since it is a compile-time tool, it is able to alter the structure of the stackframe. It does this to place all buffers after pointers to prevent pointer subversion that could be used to overwrite arbitrary memory addresses. Functions are also provided protected from argument overwriting by creating local copies of pointer arguments.

ProPolice was implemented as a patch to GCC 3.x and is currently standard and enabled by default in some Unix operating systems such as, OpenBSD, DragonFly BSD and IPCop Linux distribution. It is also standard in Gentoo Linux, but here the protection is not turned on by default. An unintrusive implementation is included in the GCC 4.1 release.

RAD

Return Address Defender (RAD) [14] is a compiler patch that stores a copy of return addresses in safe areas that it creates, and inserts necessary safeguarding code into function prologues and epilogues to defend against buffer overflow attacks. It does not require a change in the structure of stackframes, and so binary code generated by it is compatible with other existing object files and libraries. The copy of the return addresses are stored in a region of the data segment called Return Address Repository (RAR), the incorruptibility of which is ensured by marking surrounding regions as read-only. Prior to returning from a called function, the return address on the stackframe is checked against the address in the RAR, only if they match is the return address considered safe.

RAD provides two ways to protect the return addresses in RAR:

- *Minezone RAD*: The mid-section of a global integer array is declared as RAR, and the initial and final sections are marked as read-only (minezones). This prevents overwriting the middle of the RAR, where the return addresses are stored.
- *Read-Only RAD*: The whole RAR is marked as read-only and is writable only in function prologues when a new address is to be added to it. As the RAR is set as read-only, updating it in function prologues requires adding two extra system calls to each function call, which results in a serious performance penalty.

"MineZone RAD is more efficient while Read-Only RAR is more secure" [14].

Possible address storage inconsistencies that could be caused by the system calls `setjmp()` and `longjmp()` are also dealt with correctly.

System administrators are able to detect intrusions when they happen as when an attack is detected, RAD sends out an email in real-time before it terminates the attacked program.

Programs protected by RAD undergo a deterioration in performance of between 1.01 to 1.31. It is a patch to gcc-2.95.2; further details on availability can be got from here [72].

PointGuard

PointGuard is a C compiler enhancement that seeks to protect pointers by encrypting them when they are stored in memory and decrypting them just prior to dereferencing, i.e. when they are loaded into CPU registers, where they are not threatened by overwriting, as registers are not directly addressable thru calculated addresses. Attackers will still be able to corrupt pointers stored in memory, but cannot produce predictable pointer values as they have no knowledge of the decryption key.

"PointGuard critically depends on a load/store instruction discipline" [18], where pointers are always loaded into registers before being dereferenced and operated upon.

The encryption key is generated when program execution begins, using some source of randomness such as a value from `/dev/random`. This key is then kept secret within the process's address space, but is available to other processes that share memory. Encryption is done by XOR'ing pointers against the key, and decryption is done in a similar fashion by XOR'ing again against the same key. Brute force guessing is impractical, as incorrect guesses cause the process to abort, and the new process will have a different key.

A possible way to bypass PointGuard protection is to guess the decryption key by causing the program to print out pointer information via some other means.

The encryption has been kept simple so as to avoid performance overheads and performance costs have been found to be between 0%-20%.

PointGuard is no longer available.

4.5 Kernel and Operating System Alterations

Kernel and Operating system modifications try to prevent a malicious user's injected code from executing. The focus here is not on finding and fixing exploitation of software bugs but rather on prevention and containment of exploit techniques.

To use these tools the kernel of the operating system needs to be patched with them and the system built and rebooted with the patched kernel. Minor modifications might even need to be made to some of the system configuration files. The protection offered by these tools can usually be enabled or disabled via specific flags.

Bhatkar, Sekar and DuVarney

Bhatkar et al. have implemented an *address space randomization* (ASR) technique, whereby the absolute locations as well as relative distances of all code and data objects are randomized at the start of program execution (i.e. different randomizations each time the program is run). It is implemented as a source-to-source transformation which is compatible with legacy C code. Experimental results demonstrate an average run-time overhead of about 11%. The approach has a limited goal: *"it only seeks to ensure that the results of any invalid access are unpredictable"* [76]

The approach is based on the address obfuscation [4] concept, whose goal is to obscure the location of objects in memory by rearranging their positions. The techniques [4, 28, 69, 99] that have been developed to achieve this do not provide comprehensive protection against all memory error exploits, and are vulnerable to relative-address attacks, information leakage attacks, and brute-force attacks [83]. This approach aims at protecting against all known and unknown memory error exploits.

The techniques used in achieving this randomization are sketched below.

- *Randomizing variables on the stack* – at run-time the positions of variables on the stack are continuously changed. This is done via:
  1. *Shadow stack* – arrays and structures are allocated on a different stack, thereby preventing overflows from corrupting return addresses or pointer variables. The allocation order of these buffer variables is also varied for each call.
  2. *Randomizing the base of stackframes* – to blur the location of other data on the stack, the base of the stack is randomized, and stack frames are separated by random-sized gaps.
- *Static data randomization* – the location and relative order of static variables is determined at the start of the transformed program execution. Accesses to these static variables are then converted to be carried out in an indirect manner. This is done by converting the program to only have static pointer variables that store the location of these static variables. The static pointer variables are stored in read-only memory to prevent against attacks.
- *Code randomization* – code is randomized at the function granularity level. Each function is associated with a function pointer and every function call is transformed

into an indirect call via the function pointer. By updating the function pointers to point to different locations of the function body, the order of functions can be changed in the executable. Function pointers are write-protected to prevent against attacks.

In addition to these steps, *"the base of the heap, gaps between heap allocations, and the location of functions in shared libraries"* [76] are randomized.

The tool has been built for RedHat Linux 9.0 and is shipped under GPL.

Solar Designer's Non-executable stack

Stack smashing attacks are based on overwriting a function's return address on the stack to point to some arbitrary code in the stack, which is executed when the function returns. Solar Designer's Linux kernel patch [55] makes the stack segment non-executable so the operating system will not allow instructions to be executed in the this portion of a user process's virtual address space, making buffer overflow vulnerabilities more difficult to exploit.

However, making the stack non-executable can cause certain programs (some Lisp compilers) that rely on its executability to break [100]. The non-executable stack cannot prevent buffer overflow attacks that do not use the stack to place their attack code. Attack code injected into heap-allocated or statically allocated buffers can be used as exploits. The stack patch can also be circumvented by using a return-to-libc attack [98].

The patch does not impose any performance overhead nor require program recompilation. Its use however requires the kernel of the operating system to be patched. The patch is available for Linux 2.0.x, 2.2.x and 2.4.x.

PaX

PaX [68] is a Linux kernel patch that implements non-executable stack and heap memory pages. It prevents execution of unsafe code by moderating access to memory pages and is able to do so without interfering with execution of proper code. Unlike Solar Designer's non-executable stack patch, PaX is able to protect the heap as well.

It implements least privilege protection for memory pages i.e. only data in a processes's address space that needs to be executable will be granted execute permissions.

Pax provides protection in two ways – making all writable memory pages non-executable (PAGEEXEC) and by using address space layout randomization (ASLR). It uses ASLR to randomize mmap()'ed library base addresses. This counteracts many security exploits, such as buffer overflow and return-to-libc attacks, but in effect reduces them to denial of service (DoS) attacks. It incurs a small amount of overhead [69].

Shacham et al. [83] have shown that Pax ASLR is not very effective on 32-bit architectures. The randomization that it introduces is not resistant to brute force attacks.

PaX is functional and effective on many CPU architectures, including IA-32 (x86), IA-64, Alpha, PA-RISC, PowerPC, SPARC and SPARC64.

4.6 Library Patches

Library patches help make programs that are dynamically-linked safe without requiring them to be recompiled.

After the libraries have been installed, programs will need to be either implicitly or explicitly linked to them at run-time. The loader will look for these libraries and add the relevant data from them into the process's memory space.

Libsafe - Libverify

Libsafe - Libverify [3] are methods to overcome buffer overflow attacks. Both methods are implemented as dynamically loadable libraries and have performance overhead ranges between 0% - 15%.

Libsafe replaces calls to known vulnerable library functions with a comparable version that ensures that any buffer overflows are contained within the local stackframe. It tries to automatically approximate a safe upper limit on the size of buffers by recognizing that such local buffers cannot extend beyond the end of the current stack frame; i.e. it uses the calling function's frame pointer as an upper limit for writing to stack variables. The substitute function then enforces these boundaries, preventing the return address located on the stack from being overwritten.

Libsafe does not need access to program source code, nor does it require recompilation of binaries. It however fails to provide protection against heap-based buffer overflow attacks or to programs that do not use the standard C library functions to copy into the buffer [84].

It does not substitute the standard C library; instead it relies on the loader searching for it before the standard C library, so that the safe LibSafe functions are used instead of the standard unsafe library functions. By suitably setting environment variables LibSafe can be installed as the default library.

It is implemented on Linux and the source code is available under the GNU Lesser General Public License from here [54].

Libverify inserts checks dynamically into a processes' memory to implement a scheme that verifies a function's return address before it is used. Like Libsafe it works with pre-compiled binaries. On entering a function, it copies the return address onto a return address stack (*canary stack*), which is in the heap, and on exiting the function the saved return address and the actual return address are compared. If there is a mismatch, execution is halted and an alert raised.

It however does not protect the integrity of the canary stack. Libverify does not require access to the source code of the application, and hence can be used for legacy programs as well.

ContraPolice

Heap overflows occur on buffers that are dynamically allocated on the heap, e.g. by calling functions from the `malloc()` family. ContraPolice [51], which is an extension for *libc*, attempts to protect applications from *heap smashing attacks*, by protecting memory allocated on the heap from buffer overflows.

ContraPolice places a *decoy* (analogous to a canary value) before and after the allocated memory in the heap. It also maintains a list of all memory blocks that were dynamically allocated, including their base address and size. When a new memory block is requested, besides being first allocated via `malloc()`, it is also added to the list of dynamically allocated memory blocks.

Before exiting a library function handling buffers, the address of the buffer that is currently being handled is looked up to see whether it is registered in the list of dynamically allocated memory regions. If it is, a function is invoked to check if the *decoy* values before and after the allocated memory match. If they do not an error message is printed and the execution of the program is halted. Decoy values are randomly generated.

Since ContraPolice is an extension for *libc*, it is bound to a certain *libc* implementation and (depending on the *libc*) to a platform. Currently, there is only a reference implementation for *dietlibc* (a small *libc* for Linux) available.

FormatGuard

FormatGuard [17] is a patch to *glibc* that tries to prevent format string attacks by calculating the sum of arguments that a format string expects and compares this to the sum of arguments that were actually passed. If the passed number is less than the expected number, it is assumed to be an exploit and the program is terminated.

FormatGuard implements a wrapper around calls to unsafe library functions. The wrapper parses the format string to determine if there is a difference in the expected number of arguments and number of arguments demanded for by the format string.

FormatGuard is no longer available.

Summary

Table 1 summarizes the different countermeasures. A checked vulnerability column against a specific tool does not necessarily signify that the tool completely defends against that particular vulnerability. It only suggests that the tool has some mechanism for dealing with that vulnerability. This could be either by prevention, detection, mitigation or containment.

5 Conclusions

Enforcing security at the programming language level is of prime importance to systems security, as majority of attacks seek to exploit vulnerabilities in language constructs and implementation. A vast majority of programming language researchers maintain that the most effective way to address the secure programming issues is by restricting the usage of C in security-sensitive projects due to the overwhelming evidence that it is an unsafe language and hence a bad choice. They recommend the use of the aforementioned safe languages to enforce that certain programming semantics are preserved during execution [85].

Type-safe programming languages ensure that security exploits caused by memory misuse, such as malicious code execution, are prevented. They guarantee this via a number of complementary safety properties, including *memory safety* (programs can only access intended memory locations) and *flow safety* (programs can only transfer execution-flow to relevant program points).

Even so, most current type-safe languages are not the best choice for systems programming, as they do not fulfill the demanding operational requirements required for these tasks – performance, explicit memory management and control over low-level data structures. Furthermore, porting or adapting legacy code into these safe languages could be prohibitively expensive.

Moreover when it comes to programmer productivity, experience is a major factor, switching to safe programming languages could add costs in terms of training and productivity loss (at least till programmers are comfortable with the new system).

A safe efficient language with the same data and control abstractions as C, with tools to facilitate porting, would be an acceptable alternative. Within C, a more robust standard string library that performs automatic bounds and type checking would be a great boon.

Finally, new techniques such as OS virtualization, are becoming ever more pervasive in deployed systems. By isolating applications from each other the security of the system as a whole can be increased, preventing an attacker from exploiting resources other than the ones being used by the compromised application. Therefore by taking advantage of the novel alternatives offered by modern systems, i.e. by combining virtualization with secure programming languages and/or static

Tool \ Vulnerability	Stack-based buffer overflows	Heap-Based buffer overflows	Dangling pointer dereferences	Format string Attacks	Integer errors
Splint	√	√	√	√	
ITS4	√	√		√	
RATS	√	√		√	
BOON	√	√			
Purify		√	√		
Valgrind		√	√		
Systrace	√	√	√	√	
SFI	√	√	√	√	
Program Shepherding	√	√	√	√	√
SCC	√	√	√		
Fail-Safe ANSI-C compiler	√	√	√		
StackGuard	√				
ProPolice	√				
RAD	√				
PointGuard	√	√	√	√	
Bhatkar et al.	√	√	√	√	√
Solar Designer's Stack Patch	√				
PaX	√	√	√	√	√
Libsafe - Libverify	√				
ContraPolice		√			
FormatGuard				√	

Table 1: Tools vs. Vulnerabilities they attempt to counteract

.

and dynamic analysis tools, an improved assortment of secure applications could be developed.

Acknowledgments

The authors are grateful to Michael J. Fromberger for his inputs and insightful comments. We would also like to thank Robert Brentrup and Scott Rotondo for reviewing and providing feedback on the paper. We are also grateful to Sun Microsystems for their support of this project. This paper does not necessarily represent the views of our sponsors.

References

[1] Aleph One. Smashing The Stack For Fun And Profit. *Phrack*, 7(49), November 1996. http://www.phrack.org/phrack/49/P49-14. 2

[2] Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. Efficient detection of all pointer and array access errors. In *PLDI '94: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 290–301, New York, NY, USA, 1994. ACM Press. http://doi.acm.org/10.1145/178243.178446. 16

[3] A. Baratloo, N. Singh, and T. Tsai. Transparent run-time defense against stack smashing attacks. In *Proc. of the 2000 Usenix Annual Technical Conference*, Jun 2000. http://www.usenix.org/publications/library/proceedings/usenix2000/general/full_papers/baratloo/baratloo.pdf. 20

[4] S. Bhatkar, D.C. DuVarney, and R. Sekar. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *Proc. of the 12th Usenix Security Symposium*, Aug 2003. http://www.usenix.org/events/sec03/tech/full_papers/bhatkar/bhatkar.pdf. 19

[5] blexim. Basic Integer Overflows. *Phrack*, 11(60), December 2002. http://www.phrack.org/phrack/60/p60-0x0a.txt. 3

[6] Hans Boehm and Mark Weiser. Garbage Collection in an Uncooperative Environment. *Software Practice and Experience*, pages 807–820, September 1988. 4

[7] BOON - Buffer Overrun detectiON. http://www.cs.berkeley.edu/~daw/boon/. 12

[8] Derek Bruening, Evelyn Duesterwald, and Saman Amarasinghe. Design and Implementation of a Dynamic Optimization Framework for Windows. In *ACM Workshop on Feedback-Directed and Dynamic Optimization*, Austin, Texas, December 2001. http://cag.lcs.mit.edu/commit/papers/01/RIO-FDDO.pdf. 16

[9] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. An Infrastructure for Adaptive Dynamic Optimization. In *International Symposium on Code Generation and Optimization*, San Francisco, March 2003. http://cag.lcs.mit.edu/commit/papers/03/RIO-adaptive-CGO03.pdf. 16

[10] Bulba and Kil3r. Bypassing Stackguard and Stackshield. *Phrack*, 10(56), May 2000. http://www.phrack.org/phrack/56/p56-0x05. 2

[11] C− >Haskell, An Interface Generator for Haskell. http://www.cse.unsw.edu.au/~chak/haskell/c2hs/. 8

[12] CCured Documentation. http://manju.cs.berkeley.edu/ccured/. 4

[13] CERT/CC Statistics 1988-2005. http://www.cert.org/stats/cert_stats.html. 1

[14] Tzi cker Chiueh and Fu-Hau Hsu. RAD: A Compile-Time Solution to Buffer Overflow Attacks. In *ICDCS*, pages 409–417, 2001. http://www.computer.org/proceedings/icdcs/1077/10770409abs.htm. 18

[15] Jeremy Condit, Matthew Harren, Scott McPeak, George C. Necula, and Westley Weimer. CCured in the real world. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 232–244, New York, NY, USA, 2003. ACM Press. http://doi.acm.org/10.1145/781131.781157. 5

[16] Matt Conover. w00w00 on Heap Overflows. http://www.w00w00.org/files/articles/heaptut.txt, 1999. 2

[17] C. Cowan, M. Barringer, S. Beattie, G. Kroah-Hartman, M. Frantzen, and J. Lokier. FormatGuard: Automatic protection from printf format string vulnerabilities. In *Proc. of the 10th Usenix Security Symposium*, Aug 2001. http://www.usenix.org/events/sec01/full_papers/cowanbarringer/cowanbarringer.pdf. 3, 21

[18] C. Cowan, S. Beattie, J. Johansen, and P. Wagle. PointGuard™: Protecting pointers from buffer overflow vulnerabilities. In *Proc. of the 12th Usenix Security Symposium*, Aug 2003. http://www.usenix.org/events/sec03/tech/full_papers/cowan/cowan.pdf. 19

[19] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proc. of the 7th Usenix Security Symposium*, pages 63–78, Jan 1998. http://www.usenix.org/publications/library/proceedings/sec98/full_papers/cowan/cowan.pdf. 17

[20] C. Cowan, F. Wagle, C. Pu, S. Beattie, and J. Walpole. Buffer overflows: Attacks and defenses for the vulnerability of the decade. In *DARPA Information Survivability Conference & Exposition – Volume 2*, pages 119–129, Jan 2000. 10

[21] Cyclone - The Language. http://cyclone.thelanguage.org/. 5

[22] Cpogramming.com – Writing Secure Code. http://cprogramming.com/tutorial/secure.html. 2

[23] The DynamoRIO Collaboration. http://www.cag.lcs.mit.edu/dynamorio/. 16

[24] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole Jr. Exokernel: an operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, pages 251–266, Copper Mountain Resort, Colorado, December 1995. 16

[25] Hiroaki Etoh and Kunikazu Yoda. Protecting from stack-smashing attacks. http://www.research.ibm.com/trl/projects/security/ssp/main.html, June 2000. 18

[26] David Evans and David Larochelle. Improving Security Using Extensible Lightweight Static Analysis. *IEEE Softw.*, 19(1):42–51, 2002. http://dx.doi.org/10.1109/52.976940. 11

[27] About Fail-Safe C. http://web.yl.is.s.u-tokyo.ac.jp/~oiwa/FailSafe-C.html. 17

[28] S. Forrest, A. Somayaji, and D.H. Ackley. Building diverse computer systems. In *Proc. of the 6th IEEE Workshop on Hot Topics in Operating Systems*, pages 67–72, 1997. http://www.cs.unm.edu/~immsec/publications/hotos-97.pdf. 19

[29] Fortify Extra - A Taxonomy of Software Security Errors. http://vulncat.fortifysoftware.com/. 3

[30] Fortify Source Code Analysis Suite. http://www.fortifysoftware.com/products/sca.jsp. 13

[31] Sudhakar Govindavajhala and Andrew W. Appel. Using Memory Errors to Attack a Virtual Machine. In *SP '03: Proceedings of the 2003 IEEE Symposium on Security and Privacy*, page 154, Washington, DC, USA, 2003. IEEE Computer Society. http://www.cs.princeton.edu/sip/pub/memerr.pdf. 8

[32] GreenCard: A Haskell FFI Preprocessor. http://haskell.org/greencard/. 8

[33] Dan Grossman, J. Gregory Morrisett, Trevor Jim, Michael W. Hicks, Yanling Wang, and James Cheney. Region-Based Memory Management in Cyclone. In *PLDI*, pages 282–293, 2002. http://doi.acm.org/10.1145/512529.512563. 5

[34] Haskell. http://www.haskell.org/haskellwiki/Haskell. 8

[35] The Haskell 98 Foreign Function Interface 1.0: An Addendum to the Haskell 98 Report. http://www.cse.unsw.edu.au/~chak/haskell/ffi/. 8

[36] The Haskell FFI Binding Modules Generator (HSFFIG). http://hsffig.sourceforge.net/. 8

[37] HaskellDirect. http://haskell.org/hdirect/. 8

[38] R. Hastings and B. Joyce. Purify: Fast Detection of Memory Leaks and Access Errors. In *Proc. of the Winter 1992 USENIX Conference*, pages 125–138, San Francisco, California, 1991. 14

[39] Jason Hickey. Introduction to the Objective Caml Programming Language. http://www.cs.caltech.edu/courses/cs134/cs134b/book.pdf. 7

[40] IBM Rational Purify. http://www-306.ibm.com/software/awdtools/purify/. 14

[41] Igor Dobrovitski. Exploit for CVS double free() for Linux pserver, February 2003. http://seclists.org/lists/bugtraq/2003/Feb/0042.html. 2

[42] ITS4 - Software Security Tool. http://www.cigital.com/its4/. 11

[43] Daniel Jackson and Martin C. Rinard. Software Analysis: A Roadmap. In *ICSE - Future of SE Track*, pages 133–145, 2000. http://doi.acm.org/10.1145/336512.336545. 13

[44] White Paper - The Java Language Environment. http://java.sun.com/docs/white/langenv/Intro.doc.html#318. 8

[45] JavaTM 2 Platform Security Architecture. http://java.sun.com/j2se/1.5.0/docs/guide/security/spec/security-spec.doc.html. 8

[46] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *Proc. of the 2002 Usenix Annual Techincal Conference*, pages 275–288, Jun 2002. http://www.research.att.com/projects/cyclone/papers/cyclone-safety.pdf. 5, 6

[47] Vladimir Kiriansky, Derek Bruening, and Saman Amarasinghe. Secure Execution Via Program Shepherding. In *Proc. of the 11th Usenix Security Symposium*, San Francisco, August 2002. http://cag.lcs.mit.edu/commit/papers/02/RIO-security-usenix.pdf. 16

[48] Vladimir Kiriansky, Derek Bruening, and Saman Amarasinghe. Execution Model Enforcement Via Program Shepherding. MIT/LCS Technical Memo MIT/LCS Technical Memo LCS-TM-638, Massachusetts Institute of Technology, Cambridge, MA, May 2003. http://cag.lcs.mit.edu/commit/papers/03/RIO-security-TM-638.pdf. 16

[49] klog. The Frame Pointer Overwrite. *Phrack*, 9(55), Sep 1999. http://www.phrack.org/phrack/55/P55-08. 2, 18

[50] Eugene E. Kohlbecker. Syntactic Extensions in the Programming Language Lisp, 1986. 10

[51] Andreas Krennmair. ContraPolice: a libc Extension for Protecting Applications from Heap-Smashing Attacks. http://synflood.at/contrapolice.html, Nov 2003. 21

[52] David Larochelle and David Evans. Statically detecting likely buffer overflow vulnerabilities. In *10th USENIX Security Symposium*, pages 177–190. University of Virginia, Department of Computer Science, USENIX Association, August 2001. http://www.usenix.org/events/sec01/larochelle.html. 11

[53] Peng Li. Safe Systems Programming Languages, Oct 2004. http://www.seas.upenn.edu/~lipeng/homepage/papers/wpeii.pdf. 17

[54] Avaya Labs Research Libsafe. http://www.research.avayalabs.com/gcm/usa/en-us/initiatives/all/nsr.htm&Filter=ProjectTitle:Libsafe&Wrapper=LabsProjectDetails&View=LabsProjectDetails. 20

[55] Linux kernel patch from the Openwall Project. http://www.openwall.com/linux/README.shtml. 20

[56] Gary McGraw and John Viega. Improving host security with system call policies. http://www-128.ibm.com/developerworks/library/s-buffer-defend.html, 2000. 16

[57] The Memory Management Glossary. http://www.memorymanagement.org/glossary/d.html. 2

[58] Meta-Level Compilation. http://metacomp.stanford.edu/. 12

[59] Microsogt .NET Homepage. http://www.microsoft.com/net/default.mspx. 9

[60] Microsogt Visual Studio Development Center. http://msdn.microsoft.com/vstudio/. 9

[61] Mono Project. http://www.mono-project.com/Main_Page. 9

[62] George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. CCured: type-safe retrofitting of legacy software. *ACM Trans. Program. Lang. Syst.*, 27(3):477–526, 2005. http://doi.acm.org/10.1145/1065887.1065892. 4

[63] The Caml Language. http://caml.inria.fr/. 7

[64] The OCaml Tutorial. http://www.ocaml-tutorial.org/the_basics. 7

[65] Yutaka Oiwa. Implementation of a Fail-Safe ANSI C Compiler. http://web.yl.is.s.u-tokyo.ac.jp/~oiwa/thesis.pdf, Dec 2004. 17

[66] Yutaka Oiwa, Tatsurou Sekiguchi, Eijiro Sumii, and Akinori Yonezawa. Fail-safe ANSI-C compiler: An approach to making c programs secure. http://www.kb.ecei.tohoku.ac.jp/~sumii/pub/safe-C.pdf. 10, 17

[67] Ounce Labs – Prexis/Engine. http://www.ouncelabs.com/prexis_engine.html. 13

[68] Homepage of The PaX Team. http://pax.grsecurity.net/. 20

[69] PaX Project. The PaX project, Nov 2003. http://pax.grsecurity.net/docs/pax.txt. 19, 20

[70] Coverity Prevent. http://www.coverity.com/products/prevent.html. 12

[71] N. Provos. Improving Host Security with System Call Policies. In *Proc. of the 12th Usenix Security Symposium*, Aug 2003. http://www.usenix.org/events/sec03/tech/full_papers/provos/provos.pdf. 15

[72] RAD: A Compiler Time Solution to Buffer Overflow Attacks. http://www.ecsl.cs.sunysb.edu/RAD/index.html. 19

[73] RATS - Rough Auditing Tool for Security. http://www.securesoftware.com/resources/download_rats.html. 12

[74] Gerardo Richarte. Four different tricks to bypass StackShield and StackGuard protection. http://www2.corest.com/files/files/11/StackguardPaper.pdf, April-June 2002. 18

[75] rix. Smashing C++ VPTRs. *Phrack*, 0xa, May 2000. http://www.phrack.org/show.php?p=56&a=8. 2

[76] R. Sekar Sandeep Bhatkar and Daniel C. DuVarney. Efficient Techniques for Comprehensive Protection from Memory Error Exploits. In *Proc. of the 14th Usenix Security Symposium*, Aug 2005. http://www.seclab.cs.sunysb.edu/seclab/pubs/papers/usenix_sec05.pdf. 19, 20

[77] SCC: The Safe C Compiler. http://www.cs.wisc.edu/~austin/scc.html. 17

[78] The Scheme Programming Language. http://www.swiss.ai.mit.edu/projects/scheme/. 9

[79] Scheme Requests for Implementation. http://srfi.schemers.org/. 10

[80] Resources for the Scheme programming language. http://www.schemers.org/. 10

[81] Fred B. Schneider. Least privilege and more. *j-IEEE-SEC-PRIV*, 1(5):55–59, September/October 2003. 14

[82] Security and the Java Platform. http://java.sun.com/security/index.jsp. 8, 9

[83] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *ACM Conference on Computer and Communications Security*, pages 298–307, 2004. http://doi.acm.org/10.1145/1030083.1030124. 19, 20

[84] Istvan Simon. A Comparative Analysis of Methods of Defense against Buffer Overflow Attacks. http://www.mcs.csuhayward.edu/~simon/security/boflo.html, Jan 2001. 20

[85] Christian Skalka. Programming Languages and Systems Security. http://ieeexplore.ieee.org/iel5/8013/31002/01439509.pdf?tp=&arnumber=1439509&isnumber=31002. 21

[86] Splint - Secure Programming Lint. http://www.splint.org/. 11

[87] Splint User's Manual. http://www.splint.org/downloads/manual.pdf. 11

[88] Systrace Policy Generation. http://www.systrace.org/. 15

[89] Valgrind Home. http://valgrind.org/. 14

[90] Microsoft. Vault: a programming language for reliable systems. http://research.microsoft.com/vault/. 6

[91] J. Viega, J. T. Bloch, Y. Kohno, and G. McGraw. ITS4: A static vulnerability scanner for C and C++ code. In *ACSAC '00: Proceedings of the 16th Annual Computer Security Applications Conference*, page 257, Washington, DC, USA, 2000. IEEE Computer Society. http://www.cigital.com/papers/download/its4.pdf. 4, 11

[92] John Viega and Gary McGraw. *Building secure software: how to avoid security problems the right way*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002. 15

[93] David Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken. A First Step towards Automated Detection of Buffer Overrun Vulnerabilities. In *Network and Distributed System Security Symposium*, pages 3–17, San Diego, CA, February 2000. 1

[94] David Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken. A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities. In *NDSS*, 2000. http://www.isoc.org/isoc/conferences/ndss/2000/proceedings/039.pdf. 12

[95] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient Software-Based Fault Isolation. *ACM SIGOPS Operating Systems Review*, 27(5):203–216, December 1993. 15

[96] J. A. Whittaker and H. H. Thompson. *How to Break Software Security*. Addison Wesley, 2003. 4

[97] Wind Driver. http://www.windriver.com/alliances/newdirectory/product.html?ID=661. 12

[98] Rafal Wojtczuk. Defeating Solar Designers Non-executable Stack Patch. http://www.insecure.org/sploits/non-executable.stack.problems.html, 1998. 20

[99] J. Xu, Z. Kalbarczyk, and R.K. Iyer. Transparent runtime randomization for security. Technical Report UILU-ENG-03-2207, University of Illinois at Urbana-Champaign, May 2003. http://www.crhc.uiuc.edu/~junxu/Papers/TechReport_TRR_UILU-ENG-03-2207.pdf. 19

[100] Wouter Joosen Yves Younan and Frank Piessens. Code injection in C and C++: A Survey of Vulnerabilities and Countermeasures. http://www.fort-knox.org/CW386.pdf, Jul 2004. 1, 3, 6, 11, 16, 20