Adding Memory Reference Checks to Unsafe Languages – Existing Techniques and a Possible Improvement

Jason Rickwald

May 31, 2006

There are a number of common programming languages, C being one of them, that enforce weak or no type checking constraints and allow easy access to any memory location in a program's address space. These features make the language flexible, but also very unsafe. Programs written in such "unsafe" languages are more vulnerable to programming errors and attacks from malicious parties. Over the years a number of techniques have been developed to add some degree of safety into these languages (usually at the cost of performance). This paper presents some of the methods used to perform the safety checks as well as some ideas about how errors can be handled.

1. Introduction

Unsafe languages, particularly C and C++, are to blame for a great deal of security flaws in software today. One of the most common attacks on software, the buffer overflow, is directly related to the fact that memory references are not checked for validity. It is a simple task to write over the bounds of an array because it is a simple task to write and read freely to and from almost any memory location in a program's address space. Section 2 will go into more depth about unsafe languages and the security threats prevalent to them.

Ways have been developed to try to better deal with these security issues. This paper discusses how bounds checking, and pointer validity checks in general, are added to an unsafe language to make it more safe, with the emphasis being on C. Existing solutions are described and compared in section 3. This paper also discusses failure-oblivious bounds checking versus the standard abort-on-failure bounds checking. In failure-oblivious bounds checking, writes to bad memory locations are ignored and reads from bad memory locations are provided with a made-up value. There are some alternatives to this scheme which will also be discussed, but the general idea is that the program does not abort on a memory error. The failure-oblivious method of error handling will be discussed in section 4.

Lastly, in section 5, this paper proposes an enhancement to failure-oblivious computing. This enhancement, called Selective Awareness of Failures in C (SAFC), gives the programmer failure-obliviousness by default, but also allows him to opt-in to receive notifications of memory errors. Ideally, this would allow the programmer to handle problems more gracefully in parts of code that are critical.

2. The Problem

There are a number of unsafe programming languages available, and some are widely used (like C and C++). For this paper the term "unsafe," when applied to a programming language, means that it provides some mechanism, like weak type checking or arbitrary pointer creation, that allows for accidental or intentional (malicious) reading of or writing to memory locations in a way that could expose sensitive data or abnormally affect a running process. This paper focuses mostly on C/C++ and the buffer overflow attack, which is the most common attack to programs written in C/C++ [3].

The typical goal of a buffer overflow attack is to take control of a running program that has some degree privilege – either it is running at some higher privilege level or it is on a machine that the attacker cannot normally access. Two subgoals must be met to achieve this goal [3]. First, the attacker must ensure that useful code is in the privileged program's address space. Next, the attacker must get the privileged program to jump to that code.

"Buffer overflow" refers to the usual way of accomplishing these two subgoals. Programs that receive input need to place the input data somewhere in memory. A buffer of some fixed size is allocated in the program's address space and the input is read into that buffer. If the buffer is not an adequate size for the input data and the programmer did not put checks into the code, then the data will overflow the buffer and overwrite whatever memory was beyond the end of the buffer. This vulnerability can cause accidental failures due to errors in the program and it can be exploited by an attacker.

The first subgoal, ensuring that there is some useful code in the program's address space, may be solved by simply using existing code in the target program. The attacker would only need to know the location of this code in the program's address space, which may be easy to determine using utilities like `nm`. If there isn't code to do the desired work, then the attacker may inject code into the buffer.

The second subgoal, causing the program to jump to the desired code, can be done in a number of ways. This is normally where overflowing the buffer comes into play – the attacker is going to overwrite some adjacent memory location. If the buffer is on the stack, then the attacker can overwrite the return address for the current stack frame with the address of the code to execute. When the currently executing function returns, the program jumps to the attacker's code. This is commonly referred to as a "stack smash." Function pointers can also be overwritten by overflowing buffers anywhere in memory. When the program attempts to call the function using the pointer, it will instead execute the attacker's code. Lastly, an attacker can overwrite `longjmp` buffers [3]. C contains `setjmp` and `longjmp`, which can be used for simple checkpoint/rollback operations. These commands use a buffer in memory. An attacker can overflow an input buffer and overwrite the value of the `longjmp` buffer so that when `longjmp` is called it jumps to the attacker's code.

Finally, it should be mentioned that not all buffer overflow attacks try to affect the target program by executing malicious code. Some buffer overflow attacks will simply try to modify the target program's normal behavior in undesirable ways by overwriting non-executable variables adjacent to the buffer. For example, an attacker may overflow an input buffer of a web application to overwrite the username associated with his session. The attacker could then have access to the given user account.

3. The Solution

Attacks like the buffer overflow attack are a major security risk to the numerous programs written in unsafe languages that could potentially expose important assets like private data or high-load servers. Therefor, work has been done by companies and researchers towards mitigating this risk. This section describes a handful of the more common solutions to problems like buffer overflow attacks.

3.1 Don't Use an Unsafe Language

If unsafe languages are a problem, then start using “safe” languages. Such languages will typically do things like running the process in a virtual machine and/or interpreting the program in order to catch invalid memory accesses. Java does this. It also doesn't provide pointers, which allow arbitrary memory access. Instead, Java uses references, which are locked to a specific object in memory. Safe languages might also enforce strong type checking or do bounds checking on array accesses (both are true of Java as well).

So why don't we just leave it at “don't use unsafe languages?” One reason is that, despite tremendous improvements to the performance of programs written in unsafe languages (like the improved performance of JIT for Java), they still typically run slower than their unsafe counterparts. A small decrease in performance may not be an issue for most programs, but it may matter for some (like servers). Also, there is a very large amount of code written in unsafe languages that is infeasible to abandon or rewrite in safe languages.

3.2 Code Checks

If a common mistake in programs is to allocate a buffer but not check that the input data fits in the buffer, then one solution might be to just look for such mistakes in the code and correct them. Examining code for errors or even using the test/fix method of error detection is probably the most common way that programmers try to make their code “safe.” This is probably because it is the most natural way for people to do error checking. This method would be analogous to a writer proof reading his own document. Carrying the analogy further, the writer may then give his document to an editor for review. In the programming world, this would compare to a programmer submitting code to a code review by peers in his own organization or to outside auditing companies [3]. Also, the writer may utilize spelling and grammar checkers built into his word processor. Likewise, there are tools available for the static analysis of programs to detect errors and vulnerabilities and, in some cases, take steps to deal with these problems [2, 3, 7].

Checking code for errors and vulnerabilities has the advantage that it does not adversely affect the final performance of the running program. Unfortunately, there can be no guarantee made that the checked code is, indeed, free of all errors and vulnerabilities. Problems like buffer overflow vulnerabilities can be very subtle, and even obvious errors may go unnoticed by many sets of eyes. Static analysis tools, too, cannot point out all problems with a program. The semantics of C make it very hard to lay out exact rules of what is allowable. Therefor, more general heuristics are often used to find errors and vulnerabilities, sometimes leading to false-positives and undetected errors.

3.3 Stack Protection

Dealing with a whole class of vulnerability in a programming language, like buffer overflow vulnerabilities in C, can be an enormous task. It may be easier to focus on a smaller subclass of that vulnerability. This idea has found success in StackGuard, a compiler technique for detecting the most common form of buffer overflow attack – the stack smash [3]. StackGuard protects against stack smash attacks by placing a “canary” value on the stack between the local variables and the return address. At runtime, before returning from a function, the canary value is checked to make sure it is correct. It will likely be incorrect if it was overwritten by a buffer overflow attack.

StackGuard cannot guarantee invulnerability from stack smash attacks as there are a discrete number of values that can be used as the canary value – enough tries may strike upon the correct one. However, it does make the program less vulnerable to the most common buffer overflow attack with a small performance hit.

The idea of using canary values to check for overwrites can also be extended to protect against other buffer overflow attacks, as it is in PointGuard [3]. The idea is to place canary values next to other code pointers, like function pointers and `longjmp` buffers, and checking them before dereferencing the code pointer. This has the disadvantage of possibly causing incompatibility with existing programs and libraries as the canary values will change the size of data structures.

Canary values can also be taken further to protect other variables, not just code pointers. If we place a canary value next to every variable and check the canary before using the variable we can protect against even buffer overflow attacks that don't affect code jumps. However, checking canary values for all variables can degrade performance and memory usage. The authors of PointGuard, therefor, allow the programmer to specify which variables should get canary values. Unfortunately, this solution now has the programmer making non-portable modifications to the code and it has the programmer deciding which variables are "important" enough to get canary values (again exposing the program to human mistakes). It should also be noted again that these canary values only deter attacks. With enough determination and luck an attacker could find the correct canary value.

3.4 Memory Reference Checks

The most robust solution to a problem is to remove the root cause of that problem. If our problem is that our programming language does not check that data fits into its associated memory space, then we add in the default mechanisms that perform the check. To deal with buffer overflow attacks and programmer errors there have been a number of projects to add array bounds checking and, more generally, memory reference validity checks to C [1, 2, 3, 5, 6].

The idea behind these checks is actually fairly simple. For each pointer into memory we should associate with it the chunk of memory used to hold its data. This “chunk” is represented as a base address and an extent. When we dereference the pointer we check to see if the value of the pointer variable is between the base and extent for the pointer variable. This concept has been implemented in a variety of ways, each with their own advantages and drawbacks.

One such implementation is CCured [2]. CCured transforms programs written in C to add in memory safety. The first step that it takes is to perform static analysis on the program. Essentially, this static analysis tries to enforce strong typing on the program. The parts of the program that fail to adhere to the strong typing rules are deemed unsafe and are modified for runtime checking. The reason for this first step is to limit the performance hit incurred from runtime checks. If we can track a pointer through its usage in a program and see that it is never modified, casted, or used in pointer arithmetic, then there is no reason to do time consuming runtime checks on it.

CCured's approach to runtime bounds checking is rather unique. For most unsafe pointers the representation of the pointer is actually modified depending on how the pointer is classified during the static analysis phase. The new representation contains information like type, base, and extent. Placing this information right with the pointer makes creates a “wide pointer” on which checks can be performed quickly. The downside to this approach is that it limits compatibility with precompiled libraries that use the standard pointer representation.

This difficulty is managed in a couple of ways. The primary way is to create wrapper functions to act as the interface between the CCured program and the linked in functions. These functions contain special-purpose code for modifying pointer representations as they are passed through. CCured comes with many such wrapper functions already implemented for standard libraries, but the programmer may still need to write his own wrappers for other precompiled libraries used by the program. Another approach that is used less often is to store the pointer metadata in a separate structure. This approach further degrades performance during checking, and is therefor only used when complex structures must be passed between a CCured program and a precompiled library, avoiding the need to copy such structures between representations.

Another tool, called CRED, actually does use the approach of storing metadata separate from the pointer as its common approach [6]. This is done because the primary objective of the authors was to create a tool that could be used on any C program without needing to modify the code. A runtime table structure is used to map pointers to their associated static, heap, and stack objects. These objects are generally represented as their bases and extents in memory. The CRED compiler adds runtime code to the C programs that it is compiling for adding these objects to the table at load-time (for static variables), when functions are called (for stack variables), and when heap space is allocated (requiring modified versions of `malloc`, `realloc`, etc. be linked in). Steps are also taken to make sure that new pointers returned by precompiled libraries, which don't add object information to the table, don't cause the program to fail [1].

CRED has the advantage that it seems to work with a large number of programs without needing to modify the program [6] and it works effortlessly with precompiled libraries. However, it performs checks on all pointers, with each check being made up of rather slow lookups and comparisons. This can result in a sizable slowdown – up to 30% in some cases. CRED may benefit from first doing static analysis to determine what pointers are unsafe, as in CCured, but it doesn't appear that work was put into investigating this.

Another tool, presented in [7], also uses static analysis to determine what pointers are safe and unsafe. The program is then modified so that, at runtime, the unsafe pointers are tracked. To avoid the issues that come up when trying to store bases and extents for each object in memory, this tool actually uses a single bitmap to keep track of all locations in memory that are “appropriate” and “inappropriate.” This bitmap start out with all memory marked as inappropriate. When space in memory is allocated to the tracked pointers, that space gets marked as appropriate. Checking a pointer dereference consists of doing a quick lookup of that location in the bitmap. If it is appropriate, then it is allowed.

For the most part, this tool works a lot like CRED, but using a single bitmap makes pointer validity checks before dereference much faster. The disadvantage to this approach is that all “appropriate” areas in memory apply to all tracked variables. This means that it is still possible for inappropriate use of one pointer to modify the object of another pointer, but it is less likely (with the likelihood going up as more variables get marked as “tracked”). This tool also doesn't handle precompiled libraries as well as CRED and deals with them using wrapper functions as CCured does.

4. Failure-Oblivious Computing

Each of the aforementioned runtime tools may look for memory reference errors in different ways, but they each deal with errors in the same way – they abort the process. This course of action is the most natural because it is the safest. If the program is doing something bad, then terminate it before it can do any more bad. However, this may not be the ideal action in all cases. Some programs may want to terminate on an error, but only after performing some clean-up. Also, there are some errors in programs that are very minor and may hardly affect the execution of the program (if at all). Consider the example C program in Figure 1. When compiled and run this program runs just as expected – it prints out 1's for each `array` element and a 2 for the `var`. Yet it contains a vary common programming error. The while loop that assigns the 1's to the `array` elements writes one element past the array because of the use of the less-than-equals instead of less-than. It doesn't fail because the `var` memory is located after the array, and writing there is allowed. There is no problem because `var` does not get assigned a value until after the loop executes. A bounds-checked version of this program would abort even though there is no real problem. A complex program may contain many such inconsequential errors.

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

int main(int argc, char ** argv) {
    int array[5];
    int var;
    int i;

    for (i=0; i<=5; i++) {
        array[i] = 1;
    }

    var = 2;

    for (i=0; i<5; i++) {
        printf(“array[%d]=%d\n”, i, array[i]);
    }

    printf(“var=%d\n”, var);

    return 0;
}
```

Figure 1: A C program with an inconsequential error.

Another good example of where aborting is a bad idea is for server applications like web servers. Servers often use separate threads to handle each client session. If one thread is attacked by a malicious client, it would be better to terminate that thread instead of terminating the entire server process. This was the main reasoning behind the “failure-oblivious” bounds-checking compiler presented in [5]. This work started with the CRED compiler, which has already been discussed. It is therefor susceptible to the same slowdown. The authors modified CRED so that memory errors do not abort. Instead, writes to invalid memory locations are ignored and values are manufactured for reads from invalid memory locations. Some care was taken with the manufactured read values so that programs would hopefully not get stuck – random values were often returned, but most manufactured values were 1's and 0's (as these are the most common loop termination values).

Programs compiled with the failure-oblivious compiler will not abort due to inconsequential errors. In fact, other programming errors that would normally cause problems are now ignored. Consider Figure 1 again, but imagine the consequences of placing the assignment to `var` before the first loop. After compiling the program with a normal compiler we would see unexpected behavior. Compiling it with the failure-oblivious compiler would not show the unexpected behavior, because the write to `array[5]` is ignored.

[5] showed that the failure-oblivious approach was useful in many situations; particularly with servers. Input data formed for an attack on a server is changed so that it is simply bad input, which is handled in normal ways by the server's user input checks. Even if the input cannot be handled in a normal way, the attack was not allowed to corrupt critical structures like the call stack. Also, execution paths for servers handling client requests tend to be short, so that bad data will not propagate far and should not affect other user sessions.

There is a variation to the approach used in [5], which is also mentioned in [5], where writes to bad memory locations are not ignored, but are stored in a data structure. Later, if there is a bad read from the same memory location, then, rather than manufacturing a value, the value is taken out of the data structure. This approach takes more work, time, and space, but it is useful for programs that have come to rely on the presence of particular memory errors.

5. Selective Awareness of Failures in C

Experience and common sense tells us that black or white solutions are usually not the best. What we typically want is some shade of gray. Some examples have already been given for why aborting on every memory error may be bad. Ignoring all memory errors also can be bad for some programs. Programs with long execution paths, for example, may want to catch errors early in the execution rather than wasting time with what is probably bad data. Also, a mentioned before, some programs may want to know about the error so that they can abort, but only after doing some clean-up. There are probably numerous application-specific examples where the programmer may know of a better way to handle read and write errors. A programmer may know of a specific set of values that are best to return on bad reads, so he could modify error handling of bad reads to return one of these values instead of a random one.

It was this desire for more flexibility from "safe" C compilers that motivated work on SAFC – Selective Awareness of Failures in C. SAFC is the Spring of 2006 CPE 550 project of myself, Eric Firestone, and Jeremy Seeba. SAFC is an enhancement to the failure-oblivious CRED compiler presented in [5]. What is presented here is a description of the ideal SAFC implementation (in other words, what we would have done if given more time to work on it).

SAFC starts off as the CRED compiler, which is available today as a patch for the GNU gcc compiler [1]. This compiler is then modified as is described in [5] (with some help from the gcc documentation [4]) to make it failure-oblivious – bad writes are ignored and values are manufactured for bad reads. This is then further modified so that failure-obliviousness is only the default behavior. Mechanisms are added that allow programmer-specified error functions to be called on memory errors. Functions are then provided to the programmer for setting the error functions. They are specified in an optional header file. This header file does a check for a preprocessor variable that only exists if using the SAFC compiler. If the variable is not set, then it uses empty functions, allowing the program to still compile under other compilers. The function declarations for the ideal SAFC are shown in Figure 2. The actual CPE 550 implementation varies from what is shown due to difficulties with modifying gcc in the time that we had.

```
/*
 * The function called on write errors. Its parameters are, in order, the
 * size of the data being written, a pointer to the data being written,
 * a pointer to where the data was going to be written to, the name of
 * the variable used for the write (if available), the name of the source
 * file containing the error (if available), and the line in that source
 * file where the error is (if available).
 */
typedef void write_error_func(size_t, void*, void*, char*, char*, int);

/*
 * The function called on read errors. Its parameters are, in order, the
 * size of the data to read, a pointer to where the data was going to be
 * read from, the name of the variable used for the read (if available),
 * the name of the source file containing the error (if available), and
 * the line in that source file where the error is (if available). The
 * function returns a pointer to the data provided as the read value.
 */
typedef void* read_error_func(size_t, void*, char*, char*, int);

// the default error functionality
#define DEFAULT_ERROR_FUNC NULL

/*
 * Sets the function to call on write errors. Takes a pointer to a write
 * error function and returns the previous write error function. Give
 * NULL or DEFAULT_ERROR_FUNC to get the default write error functionality
 * (ignore write errors).
 *
write_error_func * set_write_error_func(write_error_func*);

/*
 * Sets the function to call on read errors. Takes a pointer to a read
 * error function and returns the previous read error function. Give
 * NULL or DEFAULT_ERROR_FUNC to get the default read error functionality
 * (manufacture a random read value, with 1's and 0's more likely).
 */
read_error_func * set_read_error_func(read_error_func*);
```

Figure 2: Functions available to the programmer for receiving error notifications in a program.

The library defines two kinds of functions: the write error function and the read error function. It then provides functions for setting the current read and write error functions. These functions also return a pointer to the previous error function so that a programmer can keep track of what function he is replacing and put it back when he no longer needs his particular error function (this can be seen in Figure 3). Both of these functions take a `NULL` to set the default failure-oblivious behavior. A small example is shown in Figure 3.

It is worth noting that the ideal SAFC implementation would also be able to handle multi-threaded applications. Each thread would have it's own set of read and write error functions.

```
void w_err(size_t s, void* dat, void* loc, char* var, char* file, int line) {
    log_error(var, file, line);
    shutdown_gracefully();
}

void* r_err(size_t s, void* loc, char* var, char* file, int line) {
    log_error(var, file, line);
    shutdown_gracefully();
    return NULL; // never reached
}

void critical_calculation() {
    write_error_func * prev_w_err;
    read_error_func * prev_r_err;

    prev_w_err = set_write_error_func(w_err);
    prev_r_err = set_read_error_func(r_err);

    /**************************
    * Some critical calculation
    **************************/

    set_write_error_func(prev_w_err);
    set_read_error_func(prev_r_err);
}
```

Figure 3: An example program where errors that are caught during a critical calculation are logged and cause a graceful shutdown.

6. Conclusion

Programming languages like C and C++ are widely used because of their efficiency and flexibility. However, as we have seen with C and C++, they may provide mechanisms that make them “unsafe,” posing a possible security risk to the large number of programs written in those languages.

This paper has presented one of the most common attacks to present day software – the buffer overflow attack. It has also presented a number of possible solutions to the problem, the most interesting of which is to add in runtime pointer validity checks to the language. The chosen solution, though, is highly dependent on one's resources, needs, and the degree of badness if an attack were to occur. For example, a small software vendor with only a few hundred customers will probably not have the resources to invest in securing their software. It would be particularly pointless if the software doesn't have much privilege (like personal information or root privilege level on the computer). A bigger company like Oracle, however, may want to invest money into ensuring some degree of security in their software, as it is widely used and may have privilege. However, customers may want their software to be as fast as possible, so they probably wouldn't go with a runtime check solution. Instead, they would be more likely to submit code to audits or put it through static analysis to find problem areas.

Lastly, this paper went into the advantages and disadvantages of abort-on-failure runtime pointer validity checks versus failure-oblivious. Both are good for some cases – abort-on-failure is good when errors should not be allowed to propagate far and failure-oblivious is good when errors won't travel far or are caught quickly by the program's input checks. However, neither of these approaches is good for all cases. Therefore, this paper also introduces a flexible way to handle pointer validity failures. This approach, called Selective Awareness of Failures in C, gives the programmer failure-obliviousness by default, but also allows him to specify error functions that can be called if an error occurs.

7. References

[1] Brugge, Herman. "SourceForge.Net: Boundschecking." SourceForge.Net. The Stanford SUIF Compiler Group. 11 May 2006 <http://sourceforge.net/projects/boundschecking>.
A patch for GCC that adds in bounds checking. This is a direct result of the CRED project (paper referenced). It serves as a basis for our implementation of failure-oblivious computing with opt-in error handling.

[2] Condit, Jeremy, Matthew Harren, Scott McPeak, George C. Necula, and Westley Weimer. CCured in the Real World. ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation, June 2003. 11 May 2006. <http://www.cs.berkeley.edu/~smcpeak/papers/ccured_pldi03.pdf>
Description of CCured -- a program that transforms C programs to add memory safety. CCured uses a combination of static analysis of the code to enforce strong type constraints where possible and runtime checks where this isn't possible. CCured actually modifies the representation of pointers, making it a unique but often incompatible solution.

[3] Cowan, Crispin, Perry Wagle, Calton Pu, Steve Beattie, and Jonathan Walpole. Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade. Proceedings of DARPA Information Survivability Conference and Exposition, Jan. 2000. 11 May 2006 <http://dx.doi.org/10.1109/DISCEX.2000.821514>.
Discussion and classification of buffer overflow attacks and defenses. Includes short descriptions of StackGuard and PointGuard, their buffer overflow defense mechanisms for C programs.

[4] Gnu Gcc Team. "GCC Online Documentation - GNU Project - Free Software Foundation (FSF)." GCC Home Page. 22 Apr. 2006. 11 May 2006 <http://gcc.gnu.org/onlinedocs/>.
Primary source of documentation for modifying GCC. Modification of GCC is required for our implementation of failure-oblivious computing with opt-in error handling.

[5] Rinard, Martin, Cristian Cadar, Daniel Dumitran, Daniel M. Roy, Tudor Leu, and William S. Beebee. Enhancing Server Availability and Security Through Failure-Oblivious Computing. OSDI 2004. 11 May 2006 <http://www.usenix.org/events/osdi04/tech/full_papers/rinard/rinard.pdf>.
Discussion and demonstration of "failure-oblivious computing." With this technique, bounds checking is added at compile time to an unsafe language (they use C), but rather than aborting on memory reference errors, the program is allowed to continue. For bad reads, a value is manufactured and returned. Bad writes are ignored.

[6] Ruwase, Olatunji, and Monica S. Lam. A Practical Dynamic Buffer Overflow Detector. 11th Annual Network and Distributed System Security Symposium, Feb. 2004. 11 May 2006 <http://suif.stanford.edu/papers/tunji04.pdf>.
Discussion and implementation of a practical and relatively low-overhead bounds checking compiler for C. This compiler, called CRED, is an open source patch for the GNU gcc compiler. It is the basis for the failure-oblivious compiler with opt-in error handling discussed in my paper.

[7] Yong, Suan Hsi, and Susan Horwitz. Protecting C Programs From Attacks Via Invalid Pointer Dereferences. 9th European Software Engineering Conference Held Jointly with 10th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2003. 11 May 2006 <http://pag.csail.mit.edu/reading-group/yong03protecting.pdf>.
Description of a tool for building C programs with memory protection. Static analysis is used to identify dangerous pointer dereferences. Runtime checks are then added to verify that those pointers point to "appropriate" locations in memory.