

Compiler Extensions To Catch Security Holes: Dissected

Tobias Franke
franke_t@rbg.informatik.tu-darmstadt.de

April 11, 2005

Abstract

Programming errors that lead to unreliable and insecure programs are a problem that has been around since the very beginning of computer science. Most of these errors are caught via extensive testing and debugging. However, more serious bugs are often left in the code unnoticed, because either the interaction through which they can occur is too complex to understand, given a certain time frame, or the testing phase is too expensive. This paper presents various automated methods, with a special focus on code analysis (both static and dynamic) to contain or isolate the damage.

1 Introduction

On August 11th of 2003, the rapidly spreading *Lovesan* worm (also known as *Blaster*), responsible for shutting down unpatched Windows XP machines every 60 seconds, used an error that was common in the world of programming for a long time: a buffer overflow. The massive damage that this particular worm caused with its dozen of siblings and modified copies is a reminder of what such a trivial thing like *bounds checking* can prevent if done properly.

However, in the age of the open source paradigm, the situation is more complex: verbatim copies of source snippets or imported libraries may contain bugs that will eventually contaminate ones own project (see [Kettlewell, 2003]). Different forks of the same codebase might be more or less secure, depending on the context it was modified in. Finally, to add to the already bad situation, not all programmers have the same notion for security: *additional checks will decrease performance* or *the cost to manage this or that for all input is too high* are two of many arguments that are often used against additional security and in favor of speed and memory. In the end, manually checking the code is tedious and error prone, while testing might require too much time investment. Thus tools exist which automate the process of security audits and checks to ensure that the most common errors are avoided.

This paper is based on [Ashcraft and Engler, 2002] and presents various methods to increase security aspects of source code, with a focus on compiler extensions for code analysis. The main attention lies on C code (since it is in wide use and very susceptible to most types of security errors), but the concepts can be transferred to any other language as well.

2 A brief overview of all methods

The programmer scene spawned a variety of tools and methods to enhance security of any type of program with some simple, often non-intrusive steps. These methods and tools are presented in this section. A brief discussion of the possible pros and contras is appended to each presentation.

2.1 Using language features

Memory leaks in unmanaged code often lead to programs susceptible to DOS style attacks, filling the systems memory up with garbage until the machine locks up. One can use the language features to detect such leaks: C and C++ allow the substitution of global memory allocation and deallocation functions like C's *malloc/free* or the equivalent *operator new/operator delete* in C++. This method is used by various libraries to count the *malloc/new* calls and subtract their respective *free/delete* calls, leaving a value of 0 if there was no allocated memory left undeleted. For instance, the Microsoft Visual Studio[Microsoft, 2004] C++ compiler comes with such a small library[MSDN, 2004], which is active in debug mode and outputs all memory leaks (see Figure 1). A

```

#include "stdafx.h"

#define CRTDBG_MAP_ALLOC
#include <stdlib.h>
#include <crtdbg.h>

int _tmain(int argc, _TCHAR* argv[])
{
    int *p = new int[3];
    _CrtDumpMemoryLeaks();
    return 0;
}
-----
Debug Output:
Detected memory leaks!
Dumping objects ->
{42} normal block at 0x002F10B0, 12 bytes long.
   Data: <          > CD CD CD CD CD CD CD CD CD CD CD CD
Object dump complete.

```

Figure 1: MS Visual Studio CrtDebug.

more sophisticated approach was taken in [Nettle, 2004], but the basic concepts for detecting memory leaks are the same.

Rewriting global memory-specific functions is an easy, non-intrusive task and requires very little effort to be included into already existing code. Also, one can use the current compile mode to either use the memory management facilities or not, so the final *release build* is not affected. Of course, the method is limited to the language and cannot be used to detect all kinds of problems within the code.

2.2 Replacement functions/libraries

A straightforward method is to empirically analyze abused vulnerabilities from the past and look for common patterns. By doing this, project members of the OpenBSD team discovered that most of the errors in the kernel tree were due to C string library calls (see [Miller and de Raadt, 1999]) that were either unsafe or simply used wrong. Those calls come in two flavors: the *fast* `str*()` and the *safe* `strn*()` (like `strcpy()` and `strncpy()`).

The downside of replacing unsafe functions in huge projects is obvious: If the interface has changed, the workload for rewriting the code is often too much. Simply replacing the calls automatically will eventually lead to wrong usage or code that didn't need the modification at all.

2.3 Compiler extensions

As described in [Ashcraft and Engler, 2002], bugs can be spotted through static code analysis, either by external parsers in a meta-compilation step, or by internal compiler extensions such as described in the paper. Similar compiler extensions are already available as of GCC 2.95[GCC, 2004]: *Stack-Smashing Protector*[Etoh, 2004] (formerly known as *ProPolice* or *StackGuard*) is a patch to the official distribution of GCC, which adds a new option *-fstack-protector*. By compiling C code with this option turned on, GCC will detect buffer overflows, prevent changes of return addresses and reorder variables to avoid memory corruption caused by an attacker. Additional code will be inserted to guarantee memory protection wherever necessary.

The downside of compiler extensions is mostly that they are proprietary to the compiler. Especially in cross-platform development, when one has to deal with different compilers and different architectures, the plug-ins or extensions might not be available or expose a completely new usage/syntax. For instance, the OpenBSD project has enhanced the *GCC* with one such compiler flag - *Wbounded* that will search for an extra code attribute to scan for boundary violations (see [OpenBSD, 2004]). This flag however is not available for other architectures.

2.4 Hardware protection

One of the most typical exploits abused in today's software has its roots in the *von Neumann architecture* of modern computers: executable code that was once data. The basic principle of *John von Neumann* was that the two parts of a program, namely code and data, share a common part of the computer's memory. Today's viruses and worms make use of this architectural design through buffer overflows, injecting new code to be executed under the current rights the system has granted to the binary that was attacked. To counter this misuse in a non-intrusive manner, newer CPUs support a filesystem-like convention to flag memory as either writable or executable. This flag - called *NX-Bit* - can be used by the operating system to protect parts of the memory.

Since the *NX-Bit* usage remains at kernel-level, software writers usually don't need to worry about memory protection at all. However, they cannot rely on it either, since the availability of this flag is CPU-dependent. On a different level, programs that make use of self-modifying code will run right into troubles: no virtual-machine (i.e. the one used by Java) or emulator will run properly, since these processes are likely to be killed for trying to execute memory they've just accessed for writing.

2.5 Software protection

On more exotic hardware, the *NX-Bit* might not be available. Thus typical kernel-implementations or enhancements of the *NX-Bit* functionality often provide another solution by emulating the flag. Currently, there are five major

implementations: W^X of the OpenBSD project, Exec Shield by Red Hat kernel developer Ingo Molnar, PaX by the PaX Team, the Linux NX Patch as of Linux 2.6.8 and the Microsoft implementation for Windows. Out of these five, the first four mentioned can emulate NX-Bit capabilities on CPU's where it is not available.

While the emulation of such a flag on legacy hardware is a huge advantage, implementers often struggle with speed penalties.

3 Code analysis

Hardware protection itself is an added bonus to the security of the underlying system. However, preventing the bugs in the first place eliminates the need for such protection if done properly, otherwise code will tend to become insecure, if the reliance upon such protection is too big. Catching bugs in source code is supported through code analysis, a method to automatically derive information about the behavior of a program, which comes in two flavors: static and dynamic. This section will discuss both methods in more detail.

3.1 Static code analysis

Static code analysis involves a so called pre- or meta-compilation step, in which a second *compiler* will run through the code and (depending on the implementation) will search for patterns that might be a source of concern. Simple forms of static code analysis are performed at compile time by modern compilers like *GCC* or *Microsoft Visual C++* (see Figure 2). Different implementations of static code analysis for special classes of errors have been shown in [Larochelle and Evans, 2001] and [Viega *et al.*, 2000]. While the first deals with buffer overflow analysis exclusively, the second approach uses a database of known vulnerability patterns to detect these kinds of errors in the source. Three other tools, which use a similar approach, are Microsoft's *PREfix*, *PREfast* and *Slam* (however, these tools use a hybrid technique that will be discussed in more detail later). Before a checkin is performed, all three tools can scan the code for common sources of bugs including

- wrong memory management (double free, freeing non allocated blocks etc.)
- wrong pointer management (dereferencing NULL or invalid pointers or pointers to freed memory)
- missing initialization of variables
- bounds violations (validation failure, buffer overruns and underflows)
- resource leakage

```

$ cat bounds.c
#include <stdio.h>

int main(int argc, char** argv)
{
    int a[4] = { 0, 1, 2, 3, 4 };
    return 0;
}
$ gcc bounds.c
bounds.c: In function 'main':
bounds.c:5: warning: excess elements in array initializer
bounds.c:5: warning: (near initialization for 'a')

```

Figure 2: Output of GCC after compiling malformed code.

By analyzing different code paths with all possible assumptions, these tools are able to catch bugs that would otherwise require much effort to be detected or wouldn't even be noticed at all, since all possible combinations are too complex to go through manually. The analyzing process of course needs the proper language information to detect error-patterns. The next two subsections will discuss how these patterns are acquired and used.

3.1.1 Rule-based

Instead of hardwiring known bug-patterns into databases or the scanners themselves, [Ashcraft and Engler, 2002] discusses a general framework to detect flaws in source code through an extensible scripting system called *metal*. The so called *checkers*, which are written in a special script-like language, describe a state machine that is applied to match patterns in the source code. For instance, a range checker would start by analyzing variables whose value was entered by an external source (network packet, system call or user input) and flag this variable as *tainted*. Following the variables path through the source code by intercepting all return calls etc., the range checker will notice if the variable itself is compared at some time in the code against other values for an upper or lower bound before reaching a *sink* (that is, before the variable is not being passed on any further). If the bounds check on that variable (if any one occurred) was not satisfying enough, because for instance there was only an upper bounds check on a signed value, then the state machine will reach the *error-state* and reports the exact path the tainted value has traveled. Otherwise, the variables tainted-flag is removed.

The challenges that remain beside writing the actual checker are to reduce *false positives/negatives* (detected or undetected states that are both wrong), and often to overcome language or OS specific analyzing difficulties. For example, the range checker will need to identify incoming variables from external network sources. In this case, the following method can be applied: packets are

both received or sent by filling a structure. To differentiate between incoming and outgoing packets, one can analyze the code if the structure is being read or written more often. However, this approach will eventually lead to false positives, since the usage of the structure depends more on the programmers *style* than on actual conventions.

On the other hand, the more dangerous *false negatives* need to be addressed. These are patterns that pose a threat to the security of the program, yet they haven't been detected by the checker. The most obvious type of pattern where this problem occurs is when for instance a tainted value is being passed through functions of a library. Since the source code for that library might not be available for further parsing, the value could be passed to a function of that library and be extracted by a subsequent call of another function of that library. Because the state is unknown while it resides inside the library calls, there are two possible solutions to this problem: either leave the tainted flag (which might produce false positives, because the library itself checks the incoming values), or remove the flag.

To sum up, the checkers are refined iteratively until they reach a general state with a minimum of false hits.

3.1.2 Belief-based

Section 3.1.1 discussed static code analysis, given that either a rule set exists or is created and refined manually. However, finding the appropriate rules for a system is tedious and often incomplete. To overcome this problem, [Engler *et al.*, 2001] discusses an even more general approach by automatically extracting rules from the code itself without any prior knowledge. So instead of searching and examining regular patterns, the code is now being analyzed for *beliefs* that match simple templates like $\langle b \rangle$ *must not follow* $\langle a \rangle$ or $\langle a \rangle$ *is coupled with* $\langle b \rangle$. By providing some of these templates, the code is being searched for contradictions (see figure 3 for an example, where a pointer is checked for validity *after* it has been dereferenced, implying that it must be valid). These contradictions are found by dividing all beliefs into two classes: MUST-beliefs, which are directly implied by the code, and MAY-beliefs that may or may not be a coincidence, depending on further statistical analysis. While contradictions of MUST-beliefs are flagged as an error, contradictions of MAY-beliefs need to be sorted and separated into actual beliefs and coincidences (if a certain pattern is found 999 of 1000 times in the code, it might be alright). For instance, in figure 3 there is a MUST-belief that the pointer *tty* is valid. This belief is contradicted by the statement two lines later, when it is checked for validity, thus provoking an error.

Looking at the big picture, the belief-based approach is an excellent way to deal with systems for which virtually no initial knowledge is available. Because the context in which the checks are applied is independent of the problem-space, one may find many errors in any project. This attribute is not only favorable while analyzing completely unknown programs, but also useful in big open-source projects where most contributors are only proficient in small parts of the

```

int mxser_write(struct tty_struct *tty, ...) {
    struct mxser_struct *info = tty->driver_data;
    unsigned long flags;

    if (!tty || !info->xmit_buf)
        return (0);

    ...
}

```

Figure 3: Impossible null-pointer check for tty (Source: [Engler *et al.*, 2001]).

source-tree. Of course, belief-based analysis suffers from *false positives/negatives* as well, because MAY-beliefs might be wrongly categorized when the statistics for a certain situation don't offer enough data.

3.2 Dynamic code analysis

While static code analysis does a good job at capturing even complex errors, many others can only be detected dynamically at run-time, since they involve the programs dynamic behavior. To address these errors, there are two major methods: The first one has been presented in 2.1, but due to its language-specific nature, it can't be applied to detect all types of errors.

The second approach is to *execute* the program after compilation with some possible automatically inserted code portions, and evaluate its run-time behavior. This procedure can be compared to typical debugging, but with some extra analyzing functionality. A variation of this method is taken by the technique behind *PREfix*, which is presented in [Bush *et al.*, 2000]. With this methodology, the analyzing process remains static, while the model that is used to capture bugs is dynamic. To do this, the whole analyzing process is split into two steps: First, the code is parsed and a complete syntax-tree is generated (as in the parsing step of a normal compiler). The syntax tree is used to determine caller-callee relationships and order of function calls. Second, the function-calls are either traced *bottom-up* (from a function back to the initial caller) or from the root or main function, simulating the current memory state of the *pseudo-executed* code in an isolated virtual machine that was set up for this task. Other implementations will insert additional code to monitor the current state of execution and then compile the program as usual. Because a complete map exists for all variables, arrays, functions and pointers, checks can be performed that would be impossible with static code analysis alone. These include checks for boundary violations of unknown or dynamic arrays, pointer and system call validity (for instance, open or closed file references), race conditions or wrong behavior after an exception. Especially the last case is a common source for memory leaks, open file descriptors and many other issues that fall into a completely new category that came to life after the introduction of *exceptions* in C++.

Proper handling of these situations is summarized by *exception safety*, which is not discussed in this paper. The virtual machine is also useful to simulate different operating environments: memory exhaustion can be tested under several conditions without changing the actual hardware, as well as execution speed.

Dynamic analysis isn't only used for debugging: CPU vendors offer so called *profilers* for their architectures to enhance performance in programs. These tools output listings that show the execution speed of certain operations, so a programmer is able to locate bottlenecks in the code. One such profiler is available at [AMD, 2004].

In direct contrast to static code analysis, the dynamic approach has its limitations: Depending on the implementation, not all code will be checked. If the program is executed, all possible tests need to be conducted (that is, all functionality needs to be tested) to cover the complete code. In case of *PREfix*, this is automated and not needed.

4 Summary

Code analysis can save a lot of time and it is the next generation of debugging. Instead of protecting the system against vulnerabilities through sandboxing techniques or memory protection, bugs are caught at the source. The main difference here is that in a protected environment, programmers eventually tend to write less secure code, because they feel protected by the underlying system, while with code analysis, the program itself is more reliable and secure.

This paper has shown two approaches to code analysis: The static approach is used like an enhanced parser, which matches certain *bug-patterns*. Because those patterns can, on code level, yield complex results, code analysis tools are likely to find bugs that wouldn't have been spotted by the programmer. However, the crucial part to this operation is that either these patterns are known and a rule set to detect them can be written, or it was already available. In neither case, a *belief-based* method can help to extract the rules from the code, by searching for contradictions.

The dynamic approach will *execute* the program and analyze the state while it is running. There are different implementations on how the binary is executed, some of which require the program to compile normally, and others that emulate the state of all variables through a virtual machine. This is needed to address problems that involve the runtime behavior of a program or which are too hard to capture through static analysis.

References

- [AMD, 2004] AMD. *CodeAnalyst Profiler*, 2004.
<http://www.developwithamd.com/appPartnerProg/codeanalyst/home/>.
- [Ashcraft and Engler, 2002] Ken Ashcraft and Dawson R. Engler. Using programmer-written compiler extensions to catch security holes. In *Proceed-*

- ings of the 2002 IEEE Symposium on Security and Privacy*, pages 143–159, Los Alamitos, CA, May 12–15 2002. IEEE Computer Society.
- [Bush *et al.*, 2000] William R. Bush, Jonathan D. Pincus, and David J. Sielaff. A static analyzer for finding dynamic programming errors. *Softw. Pract. Exper.*, 30(7):775–802, 2000.
- [Engler *et al.*, 2001] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as deviant behavior: a general approach to inferring errors in systems code. *SIGOPS Oper. Syst. Rev.*, 35(5):57–72, 2001.
- [Etoh, 2004] Hiroaki Etoh. *Stack Smashing Protection*, 2004. <http://www.trl.ibm.com/projects/security/ssp/>.
- [GCC, 2004] *GNU Compiler Collection*, 2004. <http://gcc.gnu.org/>.
- [Kettlewell, 2003] Richard Kettlewell. *Bugtraq: buffer overrun in zlib 1.1.4*, 2003. <http://seclists.org/lists/bugtraq/2003/Feb/0271.html>.
- [Larochelle and Evans, 2001] 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>.
- [Microsoft, 2004] Microsoft. *Visual Studio .net*, 2004. <http://msdn.microsoft.com/vstudio/>.
- [Miller and de Raadt, 1999] Todd C. Miller and Theo de Raadt. `strncpy` and `strncat` — consistent, safe, string copy and concatenation. In USENIX, editor, *Usenix Annual Technical Conference. June 6–11, 1999. Monterey, California, USA*, pages 175–178, Berkeley, CA, USA, 1999. USENIX. <http://www.openbsd.org/papers/strncpy-paper.ps>.
- [MSDN, 2004] Microsoft MSDN. *Enabling Memory Leak Detection*, 2004. <http://msdn.microsoft.com/library/en-us/vsdebug/html/vxconenablingmemoryleakdetection.asp>.
- [Nettle, 2004] Paul Nettle. *Memory Manager*, 2004. <http://www.fluidstudios.com/pub/FluidStudios/MemoryManagers/>.
- [OpenBSD, 2004] OpenBSD. *Manual pages: gcc-local*, 2004. <http://www.openbsd.org/cgi-bin/man.cgi?query=gcc-local>.
- [Viega *et al.*, 2000] 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. IEEE Computer Society, 2000.