A Comparison of Publicly Available Tools for Dynamic Buffer Overflow Prevention[*]

John Wilander and Mariam Kamkar
Dept. of Computer and Information Science, Linköpings universitet
{johwi, marka}@ida.liu.se

Abstract

The size and complexity of software systems is growing, increasing the number of bugs. Many of these bugs constitute security vulnerabilities. Most common of these bugs is the buffer overflow vulnerability. In this paper we implement a testbed of 20 different buffer overflow attacks, and use it to compare four publicly available tools for dynamic intrusion prevention aiming to stop buffer overflows. The tools are compared empirically and theoretically. The best tool is effective against only 50% of the attacks and there are six attack forms which none of the tools can handle.

Keywords: security intrusion; buffer overflow; intrusion prevention; dynamic analysis

1 Introduction

The size and complexity of software systems is growing, increasing the number of bugs. According to statistics from Coordination Center at Carnegie Mellon University, CERT, the number of reported vulnerabilities in software has increased with nearly 500% in two years [5] as shown in figure 1.

Now there is good news and bad news. The good news is that there is lots of information available on how these security vulnerabilities occur, how the attacks against them work, and most importantly how they can be avoided. The bad news is that this information apparently does not lead to fewer vulnerabilities. The same mistakes are made over and over again which, for instance, is shown in the statistics for the infamous *buffer overflow* vulnerability. David Wagner et al from University of California at Berkeley show that buffer overflows stand for about 50% of the vulnerabilities reported by CERT [35].

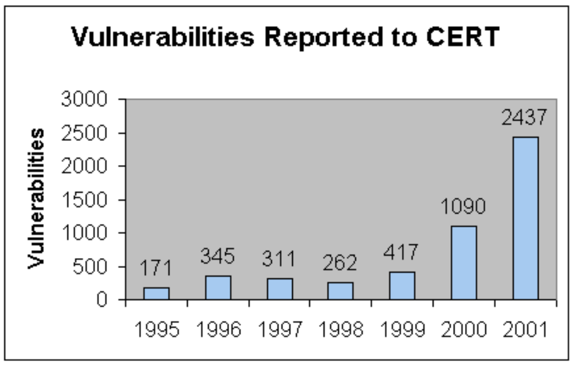

Figure 1. Software vulnerabilities reported to CERT 1995–2001.

In the middle of January 2002 the discussion about responsibility for security intrusions took a new turn. The US National Academies released a prepublication recommending that policy-makers create laws that would hold companies accountable for security breaches resulting from vulnerable products [30], which received global media attention [3, 28]. So far, only the intruder can be charged in court. In the future, software companies may be charged for not preventing intrusions. This stresses the importance of helping software engineers to produce more secure software. Automated development and testing tools aimed for security could be one of the solutions for this growing problem.

One starting point would, or could be tools that can be applied directly to the source code and solve or warn about security vulnerabilities. This means trying to solve the problems in the implementation and testing phase. Applying security related methodologies throughout the whole development cycle would most likely be more effective, but given the amount of existing software ("legacy code"), the desire for modular design using software components programmed earlier, and the time it would take to educate software engineers in secure analysis and design, we argue that security tools that aim to clean up vulnerable

[*]Paper published at 10th Network and Distributed System Security Symposium (NDSS), 2003. This work has been supported by the national computer graduate school in computer science (CUGS), commissioned by the Swedish government and the board of education.

source code are necessary. A further discussion of this issue can be found in the January/February 2002 issue of IEEE Software [18].

In this paper we investigate the effectiveness of four publicly available tools for dynamic prevention of buffer overflow attacks–namely the GCC compiler patches StackGuard, Stack Shield, and ProPolice, and the security library Libsafe/Libverify. Our approach has been to first develop an in-depth understanding of how buffer overflow attacks work and from this knowledge build a testbed with all the identified attack forms. Then the four tools are compared theoretically and empirically with the testbed. This work is a follow-up of John Wilander's Master's Thesis "Security Intrusions and Intrusion Prevention" [36].

1.1 Scope

We have tested publicly available tools for run-time prevention of buffer overflow attacks. The tools all apply to C source code, but using them requires no modifications of the source code. We do not consider approaches that use system specific features, modified kernels, or require the user to install separate run-time security components. The twenty buffer overflows represent a sample of the potential instances of buffer overflow attacks and not on the likelihood of a specific attack using the sample instance.

1.2 Paper Overview

The rest of the paper is organized as follows. Section 2 describes process memory management in UNIX and how buffer overflow attacks work. Section 3 presents the concept of intrusion prevention and describes the techniques used in the four analyzed tools. Section 4 defines our testbed of twenty attack forms and presents our theoretical and empirical comparison of the tools' effectiveness against the previously described attack forms. Section 5 describes the common shortcomings of current dynamic intrusion prevention. Finally sections 6 and 7 present related work and our conclusions.

2 Attack Methods

The analysis of intrusions in this paper concerns a subset of all violations of security policies that would constitute a security intrusion according to definitions in, for example, the Internet Security Glossary [31]. In our context an intrusion or a successful attack aims to *change the flow of control*, letting the attacker execute arbitrary code. We consider this class of vulnerabilities the worst possible since "arbitrary code" often means starting a new *shell*. This shell will have the same access rights to the system as the process attacked. If the process had *root access*, so will the attacker in the new shell, leaving the whole system open for any kind of manipulation.

2.1 Changing the Flow of Control

Changing the flow of control and executing arbitrary code involves two steps for an attacker:

1. Injecting *attack code* or *attack parameters* into some memory structure (e.g. a buffer) of the vulnerable process.

2. Abusing some vulnerable function that writes to memory of the process to alter data that controls execution flow.

Attack code could mean assembly code for starting a shell (less than 100 bytes of space will do) whereas attack parameters are used as input to code already existing in the vulnerable process, for example using the parameter `"/bin/sh"` as input to the `system()` library function would start a shell.

Our biggest concern is step two—redirecting control flow by writing to memory. That is the hard part and the possibility of changing the flow of control in this way is the most unlikely condition of the two to hold. The possibility of injecting attack code or attack parameters is higher since it does not necessarily have to violate any rules or restrictions of the program.

Changing the flow of control occurs by altering a *code pointer*. A code pointer is basically a value which gives the *program counter* a new memory address to start executing code at. If a code pointer can be made to point to attack code the program is vulnerable. The most popular target is the return address on the stack. But programmer defined *function pointers* and so called *longjmp buffers* are equally effective targets of attack.

2.2 Memory Layout in UNIX

To get a picture of the memory layout of processes in UNIX we can look at two simplified models (for a complete description see "Memory Layout in Program Execution" by Frederick Giasson [19]). Each process has a (partial) memory layout as in the figure below:

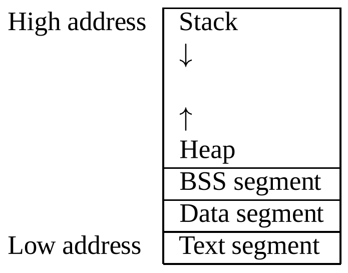

Figure 2. Memory layout of a UNIX process.

The machine code is stored in the text segment and constants, arguments, and variables defined by the programmer are stored in the other memory areas. A small C-program shows this (the comments show where each piece of data is stored in process memory):

```
static int GLOBAL_CONST = 1;      // Data segment
static int global_var;            // BSS segment

// argc & argv on stack, local
int main(argc **argv[]) {
  int local_dynamic_var;          // Stack
  static int local_static_var;    // BSS segment
  int *buf_ptr=(int *)malloc(32); // Heap
  ... }
```

For each function call a new *stack frame* is set up on top of the stack. It contains the return address, the calling function's base pointer, locally declared variables, and more. When the function ends, the return address instructs the processor where to continue executing code and the stored base pointer gives the offset for the stack frame to use.

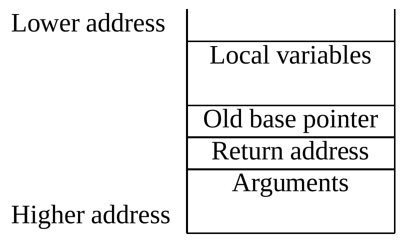

Figure 3. The UNIX stack frame.

2.3 Attack Targets

As stated above the target for a successful change of control flow is a code pointer. There are three types of code pointers to attack [11]. But Hiroaki Etoh and Kunikazu Yoda propose using the old base pointer as an attack target [15]. We have implemented their proposed attack form and proven that the old base pointer is just as dangerous a target as the return address (see section 2.4 and 4). So we have four attack targets:

1. The return address, allocated on the stack.
2. The old base pointer, allocated on the stack.
3. Function pointers, allocated on the heap, in the BSS or data segment, or on the stack either as a local variable or as a parameter.
4. Longjmp buffers, allocated on the heap, in the BSS or data segment, or on the stack either as a local variable or as a parameter.

A function pointer in C is declared as `int (*func_ptr) (char)`, in this example a pointer to a function taking a `char` as input and returns an `int`. It points to executable code.

Longjmp in C allows the programmer to explicitly jump back to functions, not going through the chain of return addresses. Let's say function A first calls `setjmp()`, then calls function B which in turn calls function C. If C now calls `longjmp()` the control is directly transferred back to function A, popping both C's and B's stack frames of the stack.

2.4 Buffer Overflow Attacks

Buffer overflow attacks are the most common security intrusion attack [35, 16] and have been extensively analyzed and described in several papers and on-line documents [29, 24, 7, 14]. Buffers, wherever they are allocated in memory, may be overflown with too much data if there is no check to ensure that the data being written into the buffer actually fits there. When too much data is written into a buffer the extra data will "spill over" into the adjacent memory structure, effectively overwriting anything that was stored there before. This can be abused to overwrite a code pointer and change the flow of control either by directly overflowing the code pointer or by first overflowing another pointer and redirect that pointer to the code pointer.

The most common buffer overflow attack is shown in the simplified example below. A local buffer allocated on the stack is overflown with 'A's and eventually the return address is overwritten, in this case with the address `0xbffff740`.

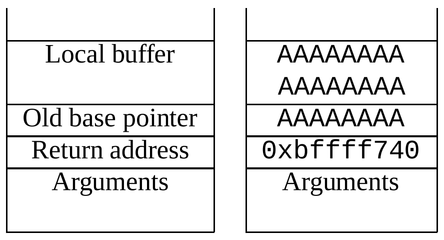

Figure 4. A buffer overflow overwriting the return address.

If an attacker can supply the input to the buffer he or she can design the data to redirect the return address to his or her attack code.

The second attack target, the old base pointer, can be abused by building a fake stack frame with a return address pointing to attack code and then overflow the buffer to overwrite the old base pointer with the address of this fake stack frame. Upon return, control will be passed to the fake stack frame which immediately returns again redirecting flow of control to the attack code.

The third attack target is function pointers. If the function pointer is redirected to the attack code the attack will be executed when the function pointer is used.

The fourth and last attack target is longjmp buffers. They contain the environment data required to resume execution from the point `setjmp()` was called. This environment data includes a base pointer and a program counter. If the program counter is redirected to attack code the attack will be executed when `longjmp()` is called.

Combining all these buffer overflow techniques, locations in memory and attack targets leaves us with no less than twenty attack forms. They are all listed in section 4 and constitute our testbed for testing of the intrusion prevention tools.

3 Intrusion Prevention

There are several ways of trying to prohibit intrusions. Halme and Bauer present a taxonomy of *anti-intrusion techniques* called *AINT* [20] where they define:

Intrusion prevention. Precludes or severely handicaps the likelihood of a particular intrusion's success.

We divide intrusion prevention into *static intrusion prevention* and *dynamic intrusion prevention*. In this section we will first describe the differences between these two categories. Secondly, we describe four publicly available tools for dynamic intrusion prevention, describe shortly how they work, and in the end compare their effectiveness against the intrusions and vulnerabilities described in section 2.4. This is not a complete survey of intrusion prevention techniques, rather a subset with the following constraints:

- Techniques used in the implementation phase of the software.
- Techniques that require no altering of source code to disarm security vulnerabilities.
- Techniques that are generic, implemented and publicly available, not prototypes or system specific tools.

Our motivation for this is to evaluate and compare tools that could easily and quickly be introduced to software developers and increase software quality from a security point of view.

3.1 Static Intrusion Prevention

Static intrusion prevention tries to prevent attacks by finding the security bugs in the source code so that the programmer can remove them. Removing all security bugs from a program is considered infeasible [23] which makes the static solution incomplete. Nevertheless, removing bugs known to be exploitable brings down the likelihood of successful attacks against all possible targets. Static intrusion prevention removes the attacker's method of entry, the security bugs. The two main drawbacks of this approach is that someone has to keep an updated database of programming flaws to test for, and since the tools only *detect* vulnerabilities the user has to know how to fix the problem once a warning has been issued.

3.2 Dynamic Intrusion Prevention

The dynamic or *run-time* intrusion prevention approach is to change the run-time environment or system functionality making vulnerable programs harmless, or at least less vulnerable. This means that in an ordinary environment the program would still be vulnerable (the security bugs are still there) but in the new, more secure environment those same vulnerabilities cannot be exploited in the same way—it protects *known* targets from attacks.

Dynamic intrusion prevention, as we will see, often ends up becoming an intrusion detection system building on program and/or environment specific solutions, terminating execution in case of an attack. The techniques are often complete in the way that they can provably secure the targets they are designed to protect (one proof can be found in a paper by Chiueh and Hsu [6]) and will produce no false positives. Their general weakness lies in the fact that they all try to solve *known* security problems, i.e. how bugs are known to be exploited today, while not getting rid of the actual bugs in the programs. Whenever an attacker has figured out a new way of exploiting a bug, these dynamic solutions often stand defenseless. On the other hand they will be effective against exploitation of any new bugs using the same attack method.

3.3 StackGuard

The *StackGuard* compiler invented and implemented by Crispin Cowan et al [10] is perhaps the most well referenced of the current dynamic intrusion prevention techniques. It is designed for detecting and stopping stack-based buffer overflows targeting the return address.

3.3.1 The StackGuard Concept

The key idea behind StackGuard is that buffer overflow attacks overwrite everything on their way towards their target. In the case of a buffer overflow on the stack targeting the return address, the attacker has to fill the buffer, then overwrite any other local variables below (i.e. on higher stack addresses), then overwrite the old base pointer until it finally reaches the return address. If we place a dummy value in between the return address and the stack data above, and then check whether this value has been overwritten or not before we allow the return address to be used, we could detect this kind of attack and possibly prevent it. The inventors have chosen to call this dummy value the *canary*.

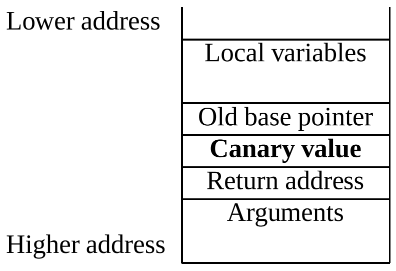

Figure 5. The StackGuard stack frame.

A potentially successful attack against such a system would be to somehow leave the canary intact while changing the return address, either by overwriting the canary with its correct value and thus not changing it, or by overwriting the return address through a pointer, not touching the canary. To solve the first problem, two canary versions have been suggested—firstly the *random canary* which consists of a random 32-bit value calculated at run-time, and secondly the *terminator canary* which consists of all four kinds of string termination sequences, namely `Null`, `Carriage Return`, `-1` and `Line Feed`. In the random canary case the attacker has to guess, or somehow retrieve, the random value at run-time. In the terminator canary case the attacker has to input all the termination sequences to keep the canary intact during the overflow. This is not possible since the string function receiving the input will terminate on one of the sequences.

Note that these techniques only stop overflow attacks that overwrite everything along the stack, not general attacks against the return address. The attacker can still abuse a pointer, making it point at the return address and writing a new address to that memory position. This shortcoming of StackGuard was discovered by Mariusz Woloszyn, alias "Emsi" and presented by Bulba and Kil3er [4]. The StackGuard team has addressed this problem by not only saving the canary value but the `XOR` of the canary and the correct return address. In this way an abused return address with an intact canary preceding it would still be detected since the `XOR` of the canary and the return address has changed. If the `XOR` scheme is used the canary has to be random since the terminator canary `XOR`ed with an address would not terminate strings anymore.

3.3.2 Random Canaries Unsupported

While testing StackGuard we noticed that the compiler did not respond to the flag set for random canary. We e-mailed Crispin Cowan and according to him: "There is only one threat that the XOR canary defeats, and the terminator canary does not: Emsi's attack. However, if you have a vulnerability that enables you to deploy Emsi's attack, then you have many other targets to attack besides function return address values. Therefore, we dropped support for random canaries [8]". We agree that the return address is not the only attack target but it is the most popular and unlike function pointers and longjmp buffers, the return address is always present. According to Cowan's e-mail and a WireX paper a better solution is on its way called *PointGuard* which will protect the integrity of pointers in general with the same kind of canary solution [11]. This implies that PointGuard will protect against all attack forms overflowing pointers (See attack forms 3a–f and 4a–f in section 4).

StackGuard is available for download at `http://www.immunix.org/`

3.4 Stack Shield

Stack Shield is a compiler patch for GCC made by Vendicator [33]. In the current version 0.7 it implements three types of protection, two against overwriting of the return address (both can be used at the same time) and one against overwriting of function pointers.

3.4.1 Global Ret Stack

The *Global Ret Stack* protection of the return address is the default choice for Stack Shield. It is a separate stack for storing the return addresses of functions called during execution. The stack is a global array of 32-bit entries. Whenever a function call is made, the return address being pushed onto the normal stack is at the same time copied into the Global Ret Stack array. When the function returns, the return address on the normal stack is replaced by the copy on the Global Ret Stack. If an attacker had overwritten the return address in one way or another the attack would be stopped without terminating the process execution. Note that no comparison is made between the return address on the stack and the copy on the Global Ret Stack. This means only prevention and no detection of an attack. The Global Ret Stack has by default 256 entries which limits the nesting depth to 256 protected function calls. Further function calls will be unprotected but execute normally.

3.4.2 Ret Range Check

A somewhat simpler but faster version of Stack Shield's protection of return addresses is the *Ret Range Check*. It uses a global variable to store the return address of the current function. Before returning, the return address on the stack is compared with the stored copy in the global variable. If there is a difference the execution is halted. Note that the Ret Range Check can detect an attack as opposed to the Global Ret Stack described above.

3.4.3 Protection of Function Pointers

Stack Shield also aims to protect function pointers from being overwritten. The idea is that function pointers normally should point into the text segment of the process' memory. That's where the programmer is likely to have implemented the functions to point at. If the process can ensure that no function pointer is allowed to point into other parts of memory than the text segment, it will be impossible for an attacker to make it point at code injected into the process, since injection of data only can be done into the data segment, the BSS segment, the heap, or the stack.

Stack Shield adds checking code before all function calls that make use of function pointers. A global variable is then declared in the data segment and its address is used as a boundary value. The checking function ensures that any function pointer about to be dereferenced points to memory below the address of the global boundary variable. If it points above the boundary the process is terminated. This protection will give false positives if the programmer has intended to use dynamically allocated function pointers.

Stack Shield is available for download at `http://www.angelfire.com/sk/stackshield/`

3.5 ProPolice

Hiroaki Etoh and Kunikazu Yoda from IBM Research in Tokyo have implemented the perhaps most sophisticated compiler protection called *ProPolice* [15].

3.5.1 The ProPolice Concept

Etoh's and Yoda's GCC patch ProPolice borrows the main idea from StackGuard (see section 3.3)—they use canary values to detect attacks on the stack. The novelty is the protection of stack allocated variables by rearranging the local variables so that `char` buffers always are allocated at the bottom, next to the old base pointer, where they cannot be overflown to harm any other local variables.

3.5.2 Building a Safe Stack Frame

After a program has been compiled with ProPolice the stack frame of functions look like that shown in figure 6.

No matter in what order local variables, pointers, and buffers are declared by the programmer, they are rearranged in stack memory to reflect the structure shown above. In this way we know that local `char` buffers can only be overflown to harm each other, the old base pointer and below. No variables can be attacked unless they are part of a `char` buffer. And by placing the canary which they call the *guard* between these buffers and the old base pointer all attacks outside the `char` buffer segment will

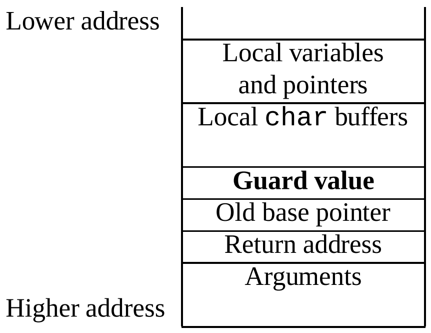

Figure 6. The ProPolice stack frame.

be detected. When an attack is detected the process is terminated.

When testing ProPolice we noticed some irregularities in when and was not the buffer overflow protection was included. It seems like small char buffers (e.g. 5 bytes) confuse ProPolice, causing it to skip the protection even if the user has set the protector flag. This gives the overall impression maybe that ProPolice is somewhat unstable.

ProPolice is available for download at `http://www.trl.ibm.com/projects/security/ssp/`

3.6 Libsafe and Libverify

Another defense against buffer overflows presented by Arash Baratloo et al [1] is *Libsafe*. This tool actually provides a combination of static and dynamic intrusion prevention. Statically it patches library functions in C that constitute potential buffer overflow vulnerabilities. A range check is made before the actual function call which ensures that the return address and the base pointer cannot be overwritten. Further protection has been provided [2] with *Libverify* using a similar dynamic approach to StackGuard (see Section 3.3).

3.6.1 Libsafe

The key idea behind Libsafe is to estimate a safe boundary for buffers on the stack at run-time and then check this boundary before any vulnerable function is allowed to write to the buffer. Vulnerable functions they consider to be the ones in table 1 below.

As a boundary value Libsafe uses the old base pointer pushed onto the stack after the return address. No local variable should be allowed to expand further down the stack than the beginning of the old base pointer. In this way a stack-based buffer overflow cannot overwrite the return address.

Function	Vulnerability
`strcpy(char *dest, const char *src)`	May overflow `dest`
`strcat(char *dest, const char *src)`	May overflow `dest`
`getwd(char *buf)`	May overflow `buf`
`gets(char *s)`	May overflow `s`
`[vf]scanf(const char *format, ...)`	May overflow arguments
`realpath(char *path, char resolved_path[])`	May overflow `path`
`[v]sprintf(char *str, const char *format, ...)`	May overflow `str`

Table 1. Vulnerable C functions that Libsafe adds protection to.

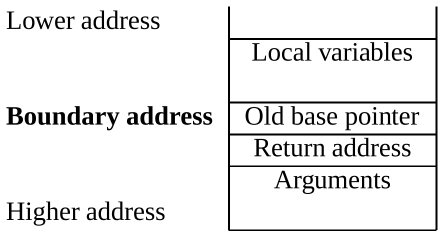

Figure 7. The Libsafe stack frame.

This boundary is enforced by overloading the functions in table 1 with wrapping functions. These wrappers first compute the length of the input as well as the allowed buffer size (i.e. from the buffer's starting point to the old base pointer) and then performs a boundary check. If the input is within the boundary the original functionality is carried out. If not the wrapper writes an alert to the system's log file and then halts the program. Observe that overflows within the local variables on the stack, such as function pointers, are not stopped.

3.6.2 Libverify

Libverify is an enhancement of Libsafe, implementing return address verification similar to StackGuard. But since this is a library it does not require recompilation of the software. As with Libsafe the library is pre-loaded and linked to any program running on the system.

The key idea behind Libverify is to alter all functions in a process so that the first thing done in every function is to copy the return address onto a *canary stack* located on the heap, and the last thing done before returning is to verify the return address by comparing it with the address saved on the canary stack. If the return address is still correct the process is allowed to continue executing. But if the return address does not match the saved copy, execution is halted and a security alert is raised. Libverify does not protect the integrity of the canary stack. They propose protecting it with `mprotect()` as in RAD (see section 3.7) but as in the RAD case this will most probably impose a very serious performance penalty [6].

To be able to do this, Libverify has to rearrange the code quite a bit. First each function is copied whole to the heap (requires executable heap) where it can be altered. Then the saving and verifying of the return address is injected into each function by overwriting the first instruction with a call to `wrapper_entry` and all return instructions with a call to `wrapper_exit`. The need for copying the code to the heap is due to the Intel CPU architecture. On other platforms this could be solved without copying the code [2].

Libverify is needed to give a more complete protection of the return address since Libsafe only addresses standard C library functions (as pointed out by Istvan Simon [32]). With Libsafe vulnerabilities could still occur where the programmer has implemented his/her own memory handling.

Libsafe and Libverify are available for download at `http://www.research.avayalabs.com/project/libsafe/`

3.7 Other Dynamic Solutions

The dynamic intrusion prevention techniques presented above are not the only ones. Other researchers have had similar ideas and implemented alternatives.

Tzi-cker Chiueh and Fu-Hau Hsu from State University of New York at Stony Brook have presented a compiler patch for protection of the return address [6]. They call their GCC patch *Return Address Defender*, or *RAD* for short. The key idea behind RAD is quite similar to the return address protection of Stack Shield described in Section 3.4. Every time a function call is made and a new stack frame is created, RAD stores a copy of the new return address. When a function returns, the return address about to be dereferenced is first checked against its copy. RAD is not publicly available.

The GCC patch *StackGhost* [25] by Mike Frantzen and Mike Shuey makes use of system specific features of the Sun Sparc Station to implement a sophisticated protection of the return address. They propose both XORing a random value with the return address (as StackGuard) as well as keeping a separate return address stack (as Stack Shield, RAD and Libverify). They also suggest using cryptographic methods instead of XOR to enhance security.

CCured and Cyclone are two recent research projects aiming to significantly enhance type and bounds checking in C. They both use a combination of static analysis and run-time checks.

CCured [27, 26] is an extension of the C programming language that distinguishes between various kinds of pointers depending on their usage. The purpose of this distinction is to be able to prevent improper usage of pointers and thus to guarantee that programs do not access memory areas they shouldn't access. CCured will change C programs slightly so that they are type safe. CCured does not change code that does not use pointers or arrays.

Cyclone [21] is a C dialect that prevents safety violations such as buffer overflows, dangling pointers, and format string attacks by ruling out certain parts of ANSI C and replacing them with safer versions. For instance `setjmp()` and `longjmp()` are unsupported (in some cases exceptions are used instead). Also pointer arithmetic is restricted. An average of 10% of the lines of code have to be changed when porting programs from C to Cyclone.

Richard Jones and Paul Kelly 1997 presented a GCC compiler patch in which they implemented run-time bounds checking of variables [22]. For each declared storage pointer they keep an entry in a table where the base and limit of the storage is kept. Before any pointer arithmetic or pointer dereferencing is made, the base and limit is checked in the table. While not explicitly aimed for security, this technique would effectively stop all kinds of buffer overflow attacks. Sadly their solution suffered both from performance penalties of more than 400 %, as well as incompatibilities with real-world programs (according to Crispin Cowan et al [9]). Because of the bad performance and compatibility we considered Jones' and Kelly's solution less interesting for software development and excluded it from our test.

It is also possible to have support for dynamic intrusion prevention in the operating system. A popular idea is the non-executable stack. This would make injection of attack code into the stack useless. But there are many ways around this protection. A few examples include using code already linked into the program from libraries (for instance calling `system()` with the parameter `"/bin/sh"`), injecting the attack code into other memory structures such as environment variables, or by exploiting buffer overflows on the heap or in the BSS/data segment. The Linux kernel patch from the Openwall Project is publicly available and implements a non-executable stack as well as protection against attacks using library functions [13]. Since it is a kernel patch it is up to the user and not the producer of software to install it. Therefore we did not include it in our test.

David Wagner and Drew Dean have presented an interesting approach for intrusion detection that relates to the functionality of the tools described in this paper [34]. They model the program's correct execution behavior via static analysis of the source code, building up callgraphs or even equivalent context-free languages defining the set of possible system call traces. Then these models are used for run-time monitoring of execution. Any deviation from the defined 'good' behavior will make the model enter an unaccepting state and trigger the intrusion alarm. As the metric for precision in intrusion detection they propose the branching factor of the model. A low branching factor means that the attacker has few choices of what to do next if he or she wants to evade detection.

4 Comparison of the Tools

Here we define our testbed of twenty buffer overflow attack forms and then present the outcome of our empirical and theoretical comparison of the tools from section 3.2.

We define an attack form as a combination of a technique, a location, and an attack target. As described in section 2.3 we have identified two techniques, two types of location and four attack targets:

Techniques. Either we overflow the buffer all the way to the attack target or we overflow the buffer to redirect a pointer to the target.

Locations. The types of location for the buffer overflow are the stack or the heap/BSS/data segment.

Attack Targets. We have four targets—the return address, the old base pointer, function pointers, and longjmp buffers. The last two can be either variables or function parameters.

Considering all practically possible combinations gives us the twenty attack forms listed below.

1. Buffer overflow on the stack all the way to the target:
   (a) Return address
   (b) Old base pointer
   (c) Function pointer as local variable
   (d) Function pointer as parameter
   (e) Longjmp buffer as local variable
   (f) Longjmp buffer as function parameter
2. Buffer overflow on the heap/BSS/data all the way to the target:
   (a) Function pointer
   (b) Longjmp buffer

Development Tool	Attacks prevented	Attacks halted	Attacks missed	Abnormal behavior
StackGuard Terminator Canary	0 (0%)	3 (15%)	16 (80%)	1 (5%)
Stack Shield Global Ret Stack	5 (25%)	0 (0%)	14 (70%)	1 (5%)
Stack Shield Range Ret Check	0 (0%)	0 (0%)	17 (85%)	3 (15%)
Stack Shield Global & Range	6 (30%)	0 (0%)	14 (70%)	0 (0%)
ProPolice	8 (40%)	2 (10%)	9 (45%)	1 (5%)
Libsafe and Libverify	0 (0%)	4 (20%)	15 (75%)	1 (5%)

Table 2. Empirical test of dynamic intrusion prevention tools. 20 attack forms tested. “Prevented” means that the process execution is unharmed. “Halted” means that the attack is detected but the process is terminated.

3. Buffer overflow of a pointer on the stack and then pointing at target:
   (a) Return address
   (b) Base pointer
   (c) Function pointer as variable
   (d) Function pointer as function parameter
   (e) Longjmp buffer as variable
   (f) Longjmp buffer as function parameter

4. Buffer overflow of a pointer on the heap/BSS/data and then pointing at target:
   (a) Return address
   (b) Base pointer
   (c) Function pointer as variable
   (d) Function pointer as function parameter
   (e) Longjmp buffer as variable
   (f) Longjmp buffer as function parameter

Note that we do not consider differences in the likelihood of certain attack forms being possible, nor current statistics on which attack forms are most popular. However, we have observed that most of the dynamic intrusion prevention tools focus on the protection of the return address. Bulba and Kil3r did not present any real-life examples of their attack forms that defeated StackGuard and Stack Shield. Also the Immunix operating system (Linux hardened with StackGuard and more) came in second place at the Defcon “Capture the Flag” competition where nearly 100 crackers and security experts tried to compromise the competing systems [12]. This implies that the tools presented here are effective against many of the currently used attack forms. The question is: will this change as soon as this kind of protection is wide spread?

Also worth noting is that just because an attack form is prevented or halted does not mean that the very same buffer overflow can not be abused in another attack form. All of these attack forms have been implemented on the Linux platform and the source code is available from our homepage: `http://www.ida.liu.se/~johwi`.

To set up the test, the source code was compiled with StackGuard, Stack Shield, or ProPolice, or linked with Libsafe/Libverify. The overall results are shown in table 2. We also made a theoretical comparison to investigate the potential of the ideas and concepts used in the tools. The overall results of the theoretical analysis are shown in table 3. For details of the tests see appendix A and B.

Most interesting in the overall test results is that the most effective tool, namely ProPolice, is able to prevent only 50% of the attack forms. Buffer overflows on the heap/BSS/data targeting function pointers or longjmp buffers are not prevented or halted by any of the tools, which means that a combination of all techniques built into one tool would still miss 30% of the attack forms.

This however does not comply with the result from the theoretical comparison. Stack Shield was not able to protect function pointers as stated by Vendicator. Another difference is the abnormal behavior of StackGuard and Stack Shield when confronted with a fake stack frame in the BSS segment.

These poor results are all evidence of the weakness in dynamic intrusion prevention discussed in section 3.2, the tested tools all aim to protect *known* attack targets. The return address has been a popular target and therefore all tools are fairly effective in protecting it.

Worth noting is that StackGuard halts attacks against the old base pointer although that was not mentioned as an explicit design goal.

Only ProPolice and Stack Shield offer real intrusion prevention—the other tools are more or less intrusion detection systems. But still the general behavior of all these tools is termination of process execution during attack.

Development Tool	Attacks prevented	Attacks halted	Attacks missed
StackGuard Terminator Canary	0 (0%)	4 (20%)	16 (80%)
StackGuard Random XOR Canary	0 (0%)	6 (30%)	14 (70%)
Stack Shield Global Ret Stack	6 (30%)	7 (35%)	7 (35%)
Stack Shield Range Ret Check	0 (0%)	10 (50%)	10 (50%)
Stack Shield Global & Range	6 (30%)	7 (35%)	7 (35%)
ProPolice	8 (40%)	3 (15%)	9 (45%)
Libsafe and Libverify	0 (0%)	6 (30%)	14 (70%)

Table 3. Theoretical comparison of dynamic intrusion prevention tools. 20 attack forms used. "Prevented" means that the process execution is unharmed. "Halted" means that the attack is detected but the process is terminated.

5 Common Shortcomings

There are several shortcomings worth discussing. We have identified four generic problems worth highlighting, especially when considering future research in this area.

5.1 Denial of Service Attacks

Since three out of four tools terminate execution upon detecting an attack they actually offer more of intrusion detection than intrusion prevention. More important is that the vulnerabilities still allow for Denial of Service attacks. Terminating a web service process is a common goal in security attacks. Process termination results in a much less serious attack but will still be a security issue.

5.2 Storage Protection

Canaries or separate return address stacks have to be protected from attacks. If the canary template or the stored copy of the return address can be tampered with, the protection is fooled. Only StackGuard with the terminator canary offers protection in this sense. The other tools have no protection implemented and the performance penalty of such protection can be very serious—up to 200 times [6].

5.3 Recompilation of Code

The three compiler patches have the common shortcoming of demanding recompilation of all code to provide protection. For software vendors shipping new products this is a natural thing but for running operating systems and legacy systems this is a serious drawback. Libsafe/Libverify offers a much more convenient solution in this sense. The StackGuard and ProPolice teams have addressed this issue by offering protected versions of Linux and FreeBSD.

5.4 Limited Nesting Depth

When keeping a separate stack with copies of return addresses, the nesting depth of the process is limited. Only Vendicator, author of Stack Shield, discusses this issue but offers no real solution to the problem.

6 Related Work

Three other studies of defenses against buffer overflow attacks have been made.

In late 2000 Crispin Cowan et al published their paper "Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade" [11]. They implicitly discuss several of our attack forms but leave out the old base pointer as an attack target. Comparison of defenses is broader considering also operating system patches, choice of programming language and code auditing but there is only a theoretical analysis, no comparative testing is done. Also the only dynamic tools discussed are their own StackGuard and their forthcoming PointGuard.

Only a month later Istvan Simon published his paper "A Comparative Analysis of Methods of Defense against Buffer Overflow Attacks" [32]. It discusses pros and cons with operating system patches, StackGuard, Libsafe, and similar solutions. The major drawback in his analysis is the lack of categorization of buffer overflow attack forms (only three of our attack forms are explicitly mentioned) and any structured comparison of the tool's effectiveness. No testing is done.

In March 2002 Pierre-Alain Fayolle and Vincent Glaume published their lengthy report "A Buffer Overflow Study, Attacks & Defenses" [17]. They describe and compare Libsafe with a non-executable stack and an intrusion detection system. Tests are performed for two of our twenty attack forms. No proper categorization of buffer overflow attack forms is made or used for testing.

7 Conclusions

There are several run-time techniques for stopping the most common of security intrusion attack—the buffer overflow. But we have shown that none of these can handle the diverse forms of attacks known today. In practice at best 40% of the attack forms were prevented and another 10% detected and halted, leaving 50% of the attacks still at large. Combining all the techniques in theory would still leave us with nearly a third of the attack forms missed. In our opinion this is due to the general weakness of the dynamic intrusion prevention solution—the tools all aim at protecting *known* attack targets, not all targets. Nevertheless these tools and the ideas they are built on are effective against many security attacks that harm software users today.

8 Acknowledgments

We are grateful to the readers who have previewed and improved our paper, especially Crispin Cowan.

References

[1] A. Baratloo, N. Singh, and T. Tsai. Libsafe: Protecting critical elements of stacks. White Paper `http://www.research.avayalabs.com/project/libsafe/`, December 1999.

[2] A. Baratloo, N. Singh, and T. Tsai. Transparent run-time defense against stack smashing attacks. In *Proceedings of the 2000 USENIX Technical Conference*, San Diego, California, USA, June 2000.

[3] L. M. Bowman. Companies on the hook for security. `http://news.com.com/2100-1023-821266.html`, January 2002.

[4] Bulba and Kil3r. Bypassig StackGuard and StackShield. Phrack Magazine 56 `http://www.phrack.org/phrack/56/p56-0x05`, May 2000.

[5] C. C. Center. Cert/cc statistics 1988-2001. `http://www.cert.org/stats/`, February 2002.

[6] T. cker Chiueh and F.-H. Hsu. RAD: A compile-time solution to buffer overflow attacks. In *Proceedings of the 21th International Conference on Distributed Computing Systems (ICDCS)*, Phoenix, Arizona, USA, April 2001.

[7] M. Conover and w00w00 Security Team. w00w00 on heap overflows. `http://www.w00w00.org/files/articles/heaptut.txt`, January 1999.

[8] C. Cowan. Personal communication, February 2002.

[9] C. Cowan, S. Beattie, R. Day, C. Pu, P. Wagle, and E. Walthinsen. Protecting systems from stack smashing attacks with StackGuard. Linux Expo `http://www.cse.ogi.edu/~crispin/`, May 1999.

[10] 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 *Proceedings of the 7th USENIX Security Conference*, pages 63–78, San Antonio, Texas, January 1998.

[11] C. Cowan, P. Wagle, C. Pu, S. Beattie, and J. Walpole. Buffer overflows: Attacks and defenses for the vulnerability of the decade. In *Proceedings of the DARPA Information Survivability Conference and Expo (DISCEX)*, pages 119–129, Hilton Head, South Carolina, January 2000.

[12] W. Crispin Cowan. Nearly 100 hackers fail to crack wirex immunix server, August 2002.

[13] S. Designer. Linux kernel patch from the openwall project. `http://www.openwall.com/linux/README`.

[14] DilDog. The tao of Windows buffer overflow. `http://www.cultdeadcow.com/cDc_files/cDc-351/`, April 1998.

[15] H. Etoh. GCC extension for protecting applications from stack-smashing attacks. `http://www.trl.ibm.com/projects/security/ssp/`, June 2000.

[16] D. Evans and D. Larochelle. Improving security using extensible lightweight static analysis. *IEEE Software*, 19(1):42–51, February 2002.

[17] P.-A. Fayolle and V. Glaume. A buffer overflow study, attacks & defenses. `http://www.enseirb.fr/~glaume/indexen.html`, March 2002.

[18] A. K. Ghosh, C. Howell, and J. A. Whittaker. Building software securely from the ground up. *IEEE Software*, 19(1):14–16, February 2002.

[19] F. Giasson. Memory layout in program execution. `http://www.decatomb.com/articles/memorylayout.txt`, October 2001.

[20] L. R. Halme and R. K. Bauer. AINT misbehaving: A taxonomy of anti-intrusion techniques. `http://www.sans.org/newlook/resources/IDFAQ/aint.htm`, April 2000.

[21] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *Proceedings of the USENIX Annual Technical Conference*, Monterey, CA, June 2002.

[22] R. Jones and P. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In *Proceedings of the Third International Workshop on Automatic Debugging AADEBUG'97*, Linkoping, Sweden, May 1997.

[23] D. Larochelle and D. Evans. Statically detecting likely buffer overflow vulnerabilities. In *Proceedings of the 2001 USENIX Security Symposium*, Washington DC, USA, August 2001.

[24] G. McGraw and J. Viega. An analysis of how buffer overflow attacks work. IBM developerWorks: Security: Security articles `http://www-106.ibm.com/developerworks/security/library/smash.html?dwzon%e=security`, March 2000.

[25] M. S. Mike Frantzen. StackGhost: Hardware facilitated stack protection. In *Proceedings of the 10th USENIX Security Symposium*, August 2001.

[26] G. Necula, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy code. In *Proceedings of the 29th Annual ACM Symposium on Principles of Programming Languages*, Portland, OR, January 2002.

[27] G. Necula, S. McPeak, and W. Weimer. Taming C pointers. In *Proceedings of ACM Conference on Programming Language Design and Implementation*, June 2002.

[28] B. News. Software security law call. http://news.bbc.co.uk/hi/english/sci/tech/newsid_1762000/1762261.stm, January 2002.

[29] A. One. Smashing the stack for fun and profit. http://immunix.org/StackGuard/profit.html, November 1996.

[30] C. Science and N. R. C. Telecommunications Board. Cybersecurity today and tomorrow: Pay now or pay later (prepublication). Technical report, National Academies, USA, http://www.nap.edu/books/0309083125/html/, January 2002.

[31] R. W. Shirey. Request for comments: 2828, Internet security glossary. http://www.faqs.org/rfcs/rfc2828.html, May 2000.

[32] I. Simon. A comparative analysis of methods of defense against buffer overflow attacks. http://www.mcs.csuhayward.edu/˜simon/security/boflo.html, January 2001.

[33] Vendicator. Stack Shield technical info file v0.7. http://www.angelfire.com/sk/stackshield/, January 2001.

[34] D. Wagner and D. Dean. Intrusion detection via static analysis. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 156–169, May 2001.

[35] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Proceedings of Network and Distributed System Security Symposium*, pages 3–17, Catamaran Resort Hotel, San Diego, California, February 2000.

[36] J. Wilander. Security intrusions and intrusion prevention. Master's thesis, Linkopings universitet, http://www.ida.liu.se/˜johwi, April 2002.

A Details of Empirical Test

Attack Target / Development Tool	Return address	Old Base Pointer	Func Ptr Variable	Func Ptr Parameter	Longjmp Buf Variable	Longjmp Buf Parameter
StackGuard Terminator Canary	Halted	Halted	Missed	Missed	Missed	Missed
Stack Shield Global Ret Stack	Prevented	Prevented	Missed	Missed	Missed	Missed
Stack Shield Range Ret Check	Abnormal	Missed	Missed	Missed	Missed	Missed
Stack Shield Global & Range	Prevented	Prevented	Missed	Missed	Missed	Missed
ProPolice	Halted	Halted	Prevented	Abnormal	Prevented	Missed
Libsafe and Libverify	Halted	Halted	Missed	Halted	Missed	Halted

Table 4. **Prevention of buffer overflow on the stack all the way to the target.**

Attack Target / Development Tool	Func Ptr Variable	Longjmp Buf Variable
StackGuard Terminator Canary	Missed	Missed
Stack Shield Global Ret Stack	Missed	Missed
Stack Shield Range Ret Check	Missed	Missed
Stack Shield Global & Range	Missed	Missed
ProPolice	Missed	Missed
Libsafe and Libverify	Missed	Missed

Table 5. **Prevention of buffer overflow on the heap/BSS/data all the way to the target.**

Attack Target / Development Tool	Return address	Old Base Pointer	Func Ptr Variable	Func Ptr Parameter	Longjmp Buf Variable	Longjmp Buf Parameter
StackGuard Terminator Canary	Missed	Halted	Missed	Missed	Missed	Missed
Stack Shield Global Ret Stack	Prevented	Prevented	Missed	Missed	Missed	Missed
Stack Shield Range Ret Check	Abnormal	Missed	Missed	Missed	Missed	Missed
Stack Shield Global & Range	Prevented	Prevented	Missed	Missed	Missed	Missed
ProPolice	Prevented	Prevented	Prevented	Prevented	Prevented	Prevented
Libsafe and Libverify	Missed	Abnormal	Missed	Missed	Missed	Missed

Table 6. **Prevention of buffer overflow of pointer on the stack and then pointing at target.**

Attack Target / Development Tool	Return address	Old Base Pointer	Func Ptr Variable	Func Ptr Parameter	Longjmp Buf Variable	Longjmp Buf Parameter
StackGuard Terminator Canary	Missed	Abnormal	Missed	Missed	Missed	Missed
Stack Shield Global Ret Stack	Prevented	Abnormal	Missed	Missed	Missed	Missed
Stack Shield Range Ret Check	Abnormal	Missed	Missed	Missed	Missed	Missed
Stack Shield Global & Range	Prevented	Prevented	Missed	Missed	Missed	Missed
ProPolice	Missed	Missed	Missed	Missed	Missed	Missed
Libsafe and Libverify	Missed	Missed	Missed	Missed	Missed	Missed

Table 7. **Prevention of buffer overflow of a pointer on the heap/BSS/data and then pointing at target.**

B Details of Theoretical Test

Attack Target	Return	Old Base	Func Ptr	Func Ptr	Longjmp Buf	Longjmp Buf
Development Tool	address	Pointer	Variable	Parameter	Variable	Parameter
StackGuard Terminator Canary	Halted	Halted	Missed	Missed	Missed	Missed
StackGuard Random XOR Canary	Halted	Halted	Missed	Missed	Missed	Missed
Stack Shield Global Ret Stack	Prevented	Prevented	Halted	Halted	Missed	Missed
Stack Shield Range Ret Check	Halted	Missed	Halted	Halted	Missed	Missed
Stack Shield Global & Range	Prevented	Prevented	Halted	Halted	Missed	Missed
ProPolice	Halted	Halted	Prevented	Missed	Halted	Missed
Libsafe and Libverify	Halted	Halted	Missed	Halted	Missed	Halted

Table 8. **Prevention of buffer overflow on the stack all the way to the target.**

Attack Target	Func Ptr	Longjmp Buf
Development Tool	Variable	Variable
StackGuard Terminator Canary	Missed	Missed
StackGuard Random XOR Canary	Missed	Missed
Stack Shield Global Ret Stack	Missed	Missed
Stack Shield Range Ret Check	Missed	Missed
Stack Shield Global & Range	Missed	Missed
ProPolice	Missed	Missed
Libsafe and Libverify	Missed	Missed

Table 9. **Prevention of buffer overflow on the heap/BSS/data all the way to the target.**

Attack Target	Return	Old Base	Func Ptr	Func Ptr	Longjmp Buf	Longjmp Buf
Development Tool	address	Pointer	Variable	Parameter	Variable	Parameter
StackGuard Terminator Canary	Missed	Halted	Missed	Missed	Missed	Missed
StackGuard Random XOR Canary	Halted	Halted	Missed	Missed	Missed	Missed
Stack Shield Global Ret Stack	Prevented	Prevented	Halted	Halted	Missed	Missed
Stack Shield Range Ret Check	Halted	Missed	Halted	Halted	Missed	Missed
Stack Shield Global & Range	Prevented	Prevented	Halted	Halted	Missed	Missed
ProPolice	Prevented	Prevented	Prevented	Prevented	Prevented	Prevented
Libsafe and Libverify	Halted	Halted	Missed	Missed	Missed	Missed

Table 10. **Prevention of buffer overflow of pointer on the stack and then pointing at target.**

Attack Target	Return	Old Base	Func Ptr	Func Ptr	Longjmp Buf	Longjmp Buf
Development Tool	address	Pointer	Variable	Parameter	Variable	Parameter
StackGuard Terminator Canary	Missed	Halted	Missed	Missed	Missed	Missed
StackGuard Random XOR Canary	Halted	Halted	Missed	Missed	Missed	Missed
Stack Shield Global Ret Stack	Prevented	Prevented	Halted	Halted	Missed	Missed
Stack Shield Range Ret Check	Halted	Halted	Halted	Halted	Missed	Missed
Stack Shield Global & Range	Prevented	Prevented	Halted	Halted	Missed	Missed
ProPolice	Missed	Halted	Missed	Missed	Missed	Missed
Libsafe and Libverify	Halted	Halted	Missed	Missed	Missed	Missed

Table 11. **Prevention of buffer overflow of a pointer on the heap/BSS/data and then pointing at target.**