RIPE: Runtime Intrusion Prevention Evaluator

John Wilander
Dept of Computer Science,
Linköpings universitet
john.wilander@gmail.com

Nick Nikiforakis
Katholieke Universiteit Leuven
Belgium
nick.nikiforakis@cs.kuleuven.be

Yves Younan
Katholieke Universiteit Leuven
Belgium
yves.younan@cs.kuleuven.be

Mariam Kamkar
Dept of Computer Science,
Linköpings universitet
mariam.kamkar@liu.se

Wouter Joosen
Katholieke Universiteit Leuven
Belgium
wouter.joosen@cs.kuleuven.be

ABSTRACT

Despite the plethora of research done in code injection countermeasures, buffer overflows still plague modern software. In 2003, Wilander and Kamkar published a comparative evaluation on runtime buffer overflow prevention technologies using a testbed of 20 attack forms and demonstrated that the best prevention tool missed 50% of the attack forms. Since then, many new prevention tools have been presented using that testbed to show that they performed better, not missing any of the attack forms. At the same time though, there have been major developments in the ways of buffer overflow exploitation.

In this paper we present *RIPE*, an extension of Wilander's and Kamkar's testbed which covers 850 attack forms. The main purpose of RIPE is to provide a standard way of testing the coverage of a defense mechanism against buffer overflows. In order to test RIPE we use it to empirically evaluate some of the newer prevention techniques. Our results show that the most popular, publicly available countermeasures cannot prevent all of RIPE's buffer overflow attack forms. ProPolice misses 60%, LibsafePlus+TIED misses 23%, CRED misses 21%, and Ubuntu 9.10 with non-executable memory and stack protection misses 11%.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection—*Invasive software*; D.2.8 [**Software Engineering**]: Metrics—*Product metric*

1. INTRODUCTION

Buffer overflows are probably the single most well-known exploitation technique in the history of computer security. The ability to take control of the execution flow of a process by overwriting adjacent data, first received world-wide attention through the Morris worm [43] and since then has been used as the exploitation mechanism in most of the well-known worms (e.g. CodeRed [29] and SQLSlammer [28]) as well as in countless attacks against popular software. Due to the severity and popularity of this attack, researchers have produced a significant amount of papers describing techniques which can, among others, detect, defend, stop and heal buffer-overflows at all possible stages of a program's lifetime. A small number of these suggested techniques have reached production level, by being included in popular programming frameworks and operating systems.

Despite the amount of research conducted in the area of buffer overflows in specific, and code injection techniques in general, modern software is still plagued by buffer-overflows which are discovered, almost weekly, in one of the many popular software products. At the time of this writing, the US-CERT [47] reports that over 200 buffer-overflow vulnerabilities have been found in 2011, many of which are in products of Microsoft, Adobe and Google. This shows that the problem of buffer overflows is far from resolved and that researchers in both academia and industry should focus their efforts in creating countermeasures that can eventually be part of real running systems.

While there are standard ways to measure the performance overhead of a buffer overflow countermeasure, such as the SPEC CPU [44] and Olden benchmarks, there is no standard way of testing and comparing the defense coverage of any given countermeasure.

In 2003, Wilander and Kamkar [53] published a comparative evaluation on runtime buffer overflow prevention using a testbed of 20 attack forms and demonstrated that the best prevention tool missed 50% of the attack forms. That testbed has been used to demonstrate the effectiveness of subsequent tools and techniques [11, 14, 15, 35, 37, 46] and the outcome of the 2003 evaluation was used to motivate further preventive research [16, 27, 36, 40].

We believe that many of the attack techniques tested by that testbed are now outdated and thus a perfect "score" of a protection countermeasure against them is of limited value.

In this paper we introduce RIPE—*Runtime Intrusion Prevention Evaluator*—which comprises of 850 buffer overflow attack forms. The main purpose of RIPE is to quantify the protection coverage of any given countermeasure by performing a wide range of buffer-overflow attacks and recording their success or failure. The tool is released as free software (see Section 10 for availability) in an attempt to standardize the comparison between countermeasures and to further support research on code-injection countermeasures. In order to test the applicability and usability of the

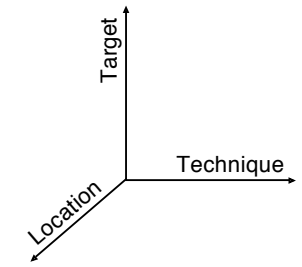
ACSAC '11 Dec. 5-9, 2011, Orlando, Florida USA

RIPE testbed, we tested it against buffer overflow countermeasures that the authors have made publicly available or that were kindly provided to us.

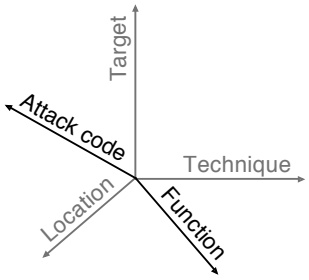
This paper and the release of the RIPE testbed provides the following research contributions:

1. Implementation of the combinatorial set of buffer overflow attack forms built on 4 locations of buffers in memory, 16 target code pointers, 2 overflow techniques, 5 variants of attack code being executed, and 10 functions being abused. In total, 850 working attack forms.
2. Empirical evaluation of publicly available buffer overflow countermeasures using canaries, boundary checking, copying and checking target data, library wrapping, and non-executable memory.
3. Open source, fully documented testbed code and driver engine for evaluation and reporting.

The rest of this paper is structured as follows: The RIPE testbed is described in Sections 2 and 3. Section 4 gives an overview of buffer overflow prevention techniques. Our evaluation setup is explained in Section 5 and the results are presented in Section 6. Section 7 contains related work. Finally, Section 8 describes future work and we conclude in Section 9.



(a) Dimensions of NDSS'03 testbed



(b) Dimensions of RIPE testbed

Figure 1: The difference of dimensions supported by the NDSS'03 testbed and RIPE

2. THE RIPE BUFFER OVERFLOW TESTBED

The RIPE (Runtime Intrusion Prevention Evaluator) testbed has a backend built in C and a frontend built in Python. The backend or "attack generator" of RIPE lets the user dynamically specify which type of buffer overflow she wants to test. For instance:

```
./ripe_attack_generator -f strcpy -t direct -l stack
-c ret -i nonop
```

... will perform the standard stack smashing attack abusing the `strcpy()` function to overflow a buffer on the stack all the way to the return pointer, redirecting it to injected attack code without a NOP sled. As another example:

```
./ripe_attack_generator -f sscanf -t indirect -l heap
-c funcptrstackparam -i returnintolibc
```

... will abuse the `sscanf()` function to perform an overflow of a buffer located on the heap, overwrite a general pointer and make it point to a function pointer parameter (indirect attack), and redirect that function pointer to return-into-libc attack code. In the next sections, we will present RIPE's *dimensions*, which are essentially all the user-configurable parameters of an attack. Then we'll enumerate all the parameters that build up the combined 850 attack forms.

2.1 Testbed Dimensions

Wilander and Kamkar's testbed (hereafter referred to as the "NDSS'03 testbed") had three dimensions all of which are included in the new RIPE testbed too; *location* of buffer in memory, *target code pointer*, and *overflow technique* - Fig 1(a).

The new RIPE testbed has five dimensions including extended versions of the original three. The additions are *attack code* and *function abused* - Fig 1(b).

2.2 Dimension 1: Location

The first testbed dimension is the memory location of the buffer to be overflowed. Both the NDSS'03 testbed and RIPE support four buffer locations; Stack, Heap, BSS, and Data segment.

2.3 Dimension 2: Target Code Pointer

The second testbed dimension is the target code pointer, i.e. the code pointer to redirect towards the attack code. RIPE supports the following target code pointers:

- *Return address*: The address stored by a function in order to return to the appropriate offset of the caller
- *Old base pointer*: The previous contents of the `EBP` register, which is used to reference function arguments and local variables
- *Function pointers*: Generic function pointers allowing programmers to dynamically call different functions from the same code
- *Longjmp buffers*: Setjmp/longjmp is a technique which allows programmers to easily jump back to a predefined point in their code (see Sec. 3.2).
- *Vulnerable Structs*: Structs which group a buffer and a function pointer and can be abused by attackers to overflow from one to the other (see Sec. 3.3).

With the exception of the *Return Address* and the *Old base pointer targets* that are Stack-specific, all other targets are allocated on all available data segments of a process, i.e. on the Stack (both as local variables and function arguments), on the Heap and on the Data/BSS segment.

2.4 Dimension 3: Overflow Technique

The third testbed dimension is overflow technique. Both the NDSS'03 testbed and RIPE support *Direct* and *Indirect* overflowing techniques. In direct techniques, the target is adjacent to the overflowed buffer or can be reached by sequentially overflowing from the buffer. On the other hand, indirect overflowing makes use of generic pointers in a two-step approach. First the generic pointer is overflowed with the address of the target and then at a later dereference, the target is overwritten with attacker-controlled

data. This technique was originally introduced by Bulba and Kil3r [10] as a way to bypass the StackGuard countermeasure. A pointer before the StackGuard canary was used to overwrite the return-address while maintaining the integrity of the canary.

2.5 Dimension 4: Attack Code

The fourth testbed dimension is new for RIPE – attack code. A user running the testbed can choose between attack code that spawns a shell on the vulnerable machine or attack code that creates a file in a specific directory. The former can be used when trying out individual attacks and the latter is used by the front-end part of RIPE which exhaustively tries all available attack combinations and then reports the full results. The variations of these two shellcodes are presented in the following list:

- *Shellcode without NOP sled*: This option can be useful in testing the accuracy of attacks as well as challenge countermeasures that rely on the detection of specific code patterns (such as the presence of a set of `0x90` bytes (NOP)) in the process' address space.
- *Shellcode with NOP sled*: This is the most-used form of shellcode that prepends the attacker's functionality with a set of NO-Operation instructions to improve the attacker's chances of correctly redirecting the execution-flow of the program into his injected code.
- *Shellcode with polymorphic NOP sled*: In this case, the NOP sled is not the standard set of `0x90` bytes but a set of instructions that can be executed without affecting the correctness of the actual attack code. As with the first variation, countermeasures that over-rely in the presence of standardized NOP-sleds will have difficulty countering such attacks. Akritidis et al. [1] conducted a study where, among others, they showed how obfuscation and encryption can be used by attackers to evade Network Intrusion Detection Systems (NIDS).
- *Return-into-libc*: Return-into-libc are attacks where the attacker does not inject new code in the process' address space but rather uses existing functions to perform his attack, for instance using the `system` libc-function to execute an interactive shell. This attack was essentially a natural evolution for attackers, when countermeasures that disallowed execution from writable memory pages, e.g. Data Execution Prevention (DEP) and $W \oplus X$, became popular in modern operating systems. RIPE uses `system()` for the spawning of an interactive shell and `creat()` for creating new files.
- *Return-Oriented Programming (ROP)*: Return-oriented programming [38] is the most recent way of carrying-out exploitation, once an attacker has achieved control of the execution flow. ROP is a generalization of Return-into-libc where now an attacker can use chunks of functionality from existing code (gadgets) and combine them to create new functionality. While we have implemented a ROP attack in our testbed, we haven't yet implemented stack-pivoting techniques and thus we can only trigger such an attack when we control the contents of the existing stack, as is the case in a stack-smashing attack.

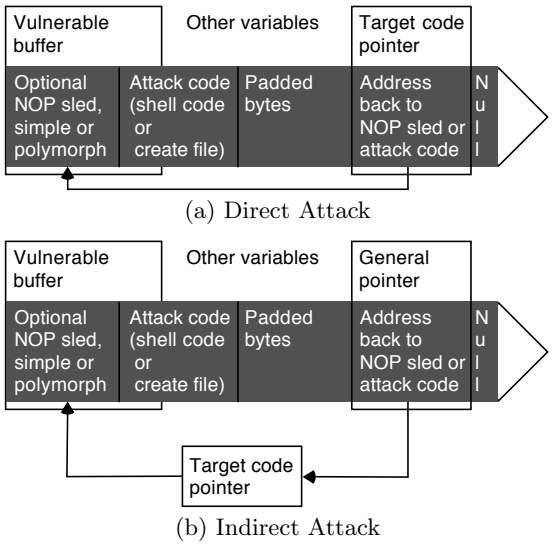


Figure 2: A direct and an indirect overflow payload with injected attack code

2.6 Dimension 5: Function Abused

The fifth and final testbed dimension is also new for RIPE – function abused. A user can choose to perform the buffer overflow with `memcpy()`, `strcpy()`, `strncpy()`, `sprintf()`, `snprintf()`, `strcat()`, `strncat()`, `sscanf()`, `fscanf()`, and also with "homebrew", a loop-based equivalent of `memcpy`.

The `n`-containing functions are designed to take the target buffer size into account which should prevent buffer overflows. The size however, is provided by the developer (static or calculated) and thus a miscalculation can cancel-out the protection offered by these functions. Known caveats include the fact that parameter `n` means *total* buffer size for `strncpy()` but *remaining* buffer space for `strncat()` [25], and if `n` is undefined for instance because of a `NULL` value in the length calculation `strncpy()` will allow for buffer overflow as shown in CVE-2009-4035 [22]:

```
line1 = getNext(line);  // May return NULL
if ((n = line1 - line) > 255) {
  n = 255;
}
strncpy(buf, line, n);  // n undef or < 0
```

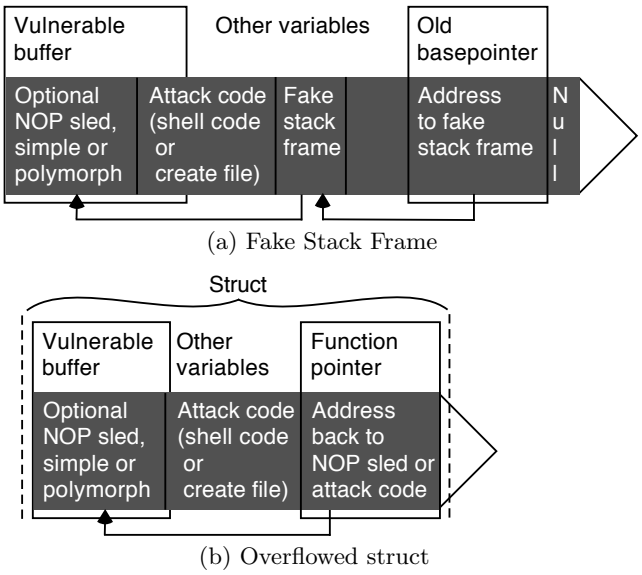
3. BUILDING PAYLOADS

RIPE's attack generator dynamically builds the specified payload and performs the attack on itself, i.e., the code contains all the required vulnerable buffers and pointers as well as the logic for offsets, attack code and overflows.

Figure 2(a) shows the payload of a direct overflow with injected code, and Figure 2(b) shows an *indirect* overflow using an intermediate pointer to target the code pointer.

3.1 Fake Stack Frame

The old base pointer is pushed on the stack immediately above the return address and is a possible target code pointer. The overflow redirects the base pointer towards an injected fake stack frame with a fake return pointer pointing to the attack code - Fig. 3(a).



(a) Fake Stack Frame

(b) Overflowed struct

Figure 3: An attack where a fake stack frame (with an attacker-controlled return address) is created and an overflow of a function pointer from within the same struct

3.2 Longjmp Buffer

Longjmp in C allows the programmer to explicitly jump back to functions, not going through the chain of return addresses. Consider a program where 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. Longjmp buffers 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.

3.3 Struct With Function Pointer

A struct containing a buffer and a function pointer can allow for an internal buffer overflow attack within the struct since there is no reordering of variables to make the code pointer unreachable from the buffer, nor are there any canary values between the buffer and the target code pointer - Fig 3(b). Such structs have been previously discussed by Zhivich et al.[54].

4. RUNTIME BUFFER OVERFLOW PREVENTION

The research in countering buffer overflow attacks at runtime has gone in several directions. We have identified six general categories or techniques, namely canary-based, boundary checking, copying and checking target data, encrypted instruction addresses, library wrappers, and non-executable and randomized memory. Our evaluation covers all but encrypted instruction addresses since we were not able to locate a publicly-available countermeasure performing such operations.

4.1 Canary-Based Tools

This technique was invented by Cowan *et al* [18] and prevents buffer overflows by adding a *canary value* to sensitive memory regions. The canary's integrity is checked before the sensitive memory is used. If the canary has been changed the sensitive memory may have been corrupted and the program is typically terminated. Other tools have adopted the canary, for instance detection of heap-based overflows targeting malloc linked lists by Robertson *et al* [36], Microsoft's `/GC` compiler flag [9], and stack protection with ProPolice by Etoh *et al* [21]. ProPolice is covered in our empirical evaluation and is presented in more detail in Sec. 5.1.

4.2 Boundary Checking Tools

Standard C and C++ do not have runtime bounds checking unlike newer languages such as Java and C#. This is one of the fundamental design decisions that make buffer overflow attacks possible. Researchers have implemented variants of C compilers that include boundary checking in binaries.

In 1997 Jones and Kelly presented a GCC compiler patch in which they implemented runtime bounds checking of variables [26]. Sadly their solution suffered from performance penalties of more than 400%, as well as incompatibility with real-world programs [17]. Ruwase and Lam continued Jones' and Kelly's work and have implemented a GCC patch called "CRED" [37]. CRED is covered in our empirical evaluation and is presented in section 5.5.

4.3 Tools Copying and Checking Target Data

StackShield [50] and Libverify [5] were the first buffer overflow prevention tools that used the technique of storing copies of return addresses on a separate stack. When a function returned, its stored return address is checked against the copy on the separate stack. If the addresses differed either the correct address was copied back or execution was halted. StackShield is a compiler patch whereas Libverify patched the code during load. Both StackShield and Libverify are covered in our empirical evaluation and are presented in section 5.2 and 5.3.

Chiueh and Hsu [13] presented a compiler patch called *RAD* in 2001. It used a separate stack to keep copies of return addresses similar to StackShield. Smirnov and Chiueh have continued the work and implemented a more complex GCC patch called *DIRA* [42]. Apart from the separate stack with copies of return addresses, DIRA keeps copies of function pointer values in a special buffer.Nebenzahl and Wool [30] have developed a technique for instrumenting Windows binaries at install-time with a separate stack for copies of return addresses.

4.4 Library Wrappers

Buffer overflow prevention through library wrappers was originally done by Baratloo, Singh, and Tsai, and their tool was called *Libsafe* [4]. It patches library functions in C that constitute potential buffer overflow vulnerabilities. In the patched functions a range check is made before the actual function call. As a boundary value Libsafe uses the old base pointer pushed onto the stack after the return address.

Avijit, Gupta and Gupta continued the work by Baratloo *et al* by implementing *LibsafePlus* and *TIED* [3, 2]. Their system collects and stores information about the sizes of both stack and heap buffers. This information is then used

at runtime to ensure that no character buffers are written past their limit. Libsafe, LibsafePlus, and TIED are all covered in our empirical evaluation and are presented in more detail in Sections 5.3 and 5.4.

4.5 Non-Executable and Randomized Memory

The Linux kernel patch from the Openwall Project was the first to implement a non-executable stack [19]. Not allowing execution of code stored on the stack effectively stops execution of attack code injected on the stack and on the heap. In some cases, researchers have been able to circumvent this countermeasure by abusing certain traits of a program, e.g. convincing the Just-in-time compiler of ActionScript in Macromedia Flash to place attacker-code in their writable and executable memory pages [8].

Two more recent kernel patches that deny execution both on the stack and on the heap are PaX [23] and ExecShield [49]. They also randomize address offsets from the base of memory locations, called *Address Space Layout Randomization*, to further countermeasure buffer overflow attacks. DieHard [6] and its continuation DieHarder [31] are memory allocators which randomize the location of heap objects on the heap and require larger-than-needed address spaces to ensure probabilistic safety. Even though DieHard is publicly available we could not include it in our evaluation since RIPE is a process that attacks itself calculating the needed offsets from within its source code (see Sec. 8). This means, that RIPE would "unfairly" de-randomize DieHard and successfully perform all of the attacks.

5. EMPIRICAL EVALUATION SETUP

We have used RIPE to evaluate a number of preventive tools and techniques designed to counter buffer overflow attacks, namely ProPolice (canary-based), CRED (boundary checking), StackShield and Libverify (copying and checking target data), Libsafe, LibsafePlus, LibsafePlus+TIED (library wrappers), and PAE and XD (non-executable memory).

The theoretical number of attack forms produced by multiplying all the choices is 3,840 (4 locations * 16 target code pointers * 2 techniques * 3 variants of attack code without NOP sled variations * 10 functions being abused). However, that number incorporates 2,990 practically impossible attack forms. For instance it is not possible to perform a direct buffer overflow all the way from the BSS segment to the stack due to the unmapped heap pages and the guard page separating the stack and heap. Thus, the number of working attack forms is 850. In the empirical evaluation we have left out the three NOP versions and only executed with one (the simple NOP sled). Since none of the protection tools or techniques evaluated tries to detect NOP sleds *per se* including them would not change the results of the current analysis. Nevertheless RIPE supports three different NOP sled settings for attack code.

5.1 ProPolice

Hiroaki Etoh and Kunikazu Yoda from IBM Research in Tokyo have implemented a compiler protection called *ProPolice* [21]. It borrows the main idea from StackGuard—canary, or *guard* values to detect attacks on the stack. The guard is placed between the buffers and the old base pointer meaning it protects both the return pointer and the old base-pointer from direct overflows. In addition to the guard, ProPolice rearranges the local stack variables so that `char` buffers always are allocated at the bottom, next to the canary, where they cannot harm any other local variables if overflowed. Non-char buffer variables can only be attacked if they are part of a struct that also contains a buffer.

5.2 StackShield

StackShield is a compiler patch for GCC made by Vendicator [50]. 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.

5.2.1 Global Ret Stack

The *Global Ret Stack* protection of the return address is the default choice for StackShield. 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 allowing the countermeasure to prevent but not to detect buffer overflows (and possible corruption of data due to them). 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.

5.2.2 Ret Range Check

A somewhat simpler but faster version of StackShield'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.

5.2.3 Protection of Function Pointers

StackShield 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 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 stack, the heap, the BSS, or the data segment.

StackShield 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 program uses dynamically allocated function pointers.

5.3 Libsafe and Libverify

Another defense against buffer overflows presented by Arash Baratloo et al [4] 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 [5] with *Libverify* using a similar dynamic approach to StackGuard.

5.3.1 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.

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. This boundary is enforced by overloading `strcpy()`, `strcat()`, `getwd()`, `gets()`, `[vf]scanf()`, `realpath()`, and `[v]sprintf()` 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.

5.3.2 Libverify

Libverify is an enhancement of Libsafe, implementing return address verification similar to StackShield. However, 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. However, 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()` like *Return Address Defender*, RAD [13]. However, as in the RAD case this will most probably impose a serious performance penalty.

To be able to do this, Libverify has to transform the code of a given program. 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 [5]. 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 [41]). With Libsafe vulnerabilities could still occur where the programmer has implemented his/her own memory handling.

5.4 LibsafePlus and TIED

Avijit, Gupta and Gupta continued the work by Baratloo *et al* by implementing *LibsafePlus* and *TIED* [3, 2]. TIED collects static information and LibsafePlus collects dynamic information about the sizes of stack and heap buffers. This information is used runtime to ensure that no character buffers are written past their limit.

Static buffer sizes are collected compile-time by exploiting debugging information produced by a specific compiler option. Dynamic buffer size information is collected runtime by interception of calls to `malloc()` and `free()`. Finally, the original technique with wrappers for dynamically linked libraries handling strings is used to check the bounds. Their main contributions are a more precise boundary check of stack buffers than the previous solution, and a boundary check of heap buffers.

5.5 CRED

Ruwase and Lam continued Jones' and Kelly's boundary checking work and have implemented a GCC patch called "CRED", *C Range Error Detector* [37]. Their goals were for the runtime checks to impose less overhead and provide better compatibility. To enhance performance they only perform boundary checks on string buffers since they consider such buffers the most likely ones vulnerable to security attacks. With such a restriction most of the programs they tested suffered less than 26 % overhead. Worst case was a string-intensive email program which suffered 130 % overhead.

Compatibility was solved by storing out-of-bounds pointer values in so called *out-of-bounds objects*. If pointer arithmetics using the out-of-bounds pointer results in an in-bounds address the pointer is sanitized. All variables with a memory range such as arrays and structs get an associated *referent object* that keeps of pointer arithmetic and bounds. Pointer operations that reference memory outside the referent object are illegal. CRED allows out-of-bounds references to be part of arithmetic as long as the resulting access is within bounds.

5.6 Non-Executable Memory and Stack Protector (Ubuntu 9.10)

We have evaluated the buffer overflow prevention techniques used in Ubuntu 9.10 "Karmic" [33] which has several security features [34] relevant to buffer overflow prevention.

5.6.1 ASLR

Ubuntu 9.10 has five ASLR (Address Space Layout Randomization) features, four enabled by default—Stack ASLR, Libs/mmap ASLR, Exec ASLR, brk ASLR, and VDSO ASLR [34].

The fundamentals of defeating ASLR have been studied by Schacham *et al* [39]. Attackers may reduce the entropy present in a randomized address space by leaking information via format string attacks [20], buffer over-reads [45] or covering multiple bits of entropy per attack by using heap spraying, introduced by Hassell and Permeh [24]. RIPE does not use brute force, information leakage, or heap spraying to circumvent ASLR. While such attack methods are inter-

esting, RIPE currently focusses on evaluating countermeasures performing attack detection or active prevention, not on countermeasures making attacks harder. RIPE calculates its offsets and target addresses at runtime and thus has information available *after* randomization. The effects of this decision are discussed in detail in Section 8.

5.6.2 Non-Executable Memory

Most modern CPUs protect against executing non-executable memory regions (heap, stack, etc), known either as Non-eXecute (NX) or eXecute-Disable (XD) [52]. This protection reduces the areas an attacker can use to perform arbitrary code execution. It requires that the kernel uses PAE, *Physical Address Extension*. Ubuntu 9.10, partially emulates this protection for processors lacking NX when running on a 32bit kernel. Our evaluation runs where performed on a machine with an Intel Core 2 Duo processor with XD support enabled.

5.6.3 Stack Protector (ProPolice)

Ubuntu 9.10 ships with a patched gcc using `-fstack-protector` by default. This protection is ProPolice, presented in section 5.1. RIPE was compiled with this gcc for the Ubuntu 9.10 evaluation.

6. EMPIRICAL EVALUATION RESULTS

In this section, we present the summary of our empirical evaluation of the protection tools and techniques presented in section 5. We then continue with detailed evaluation results for the top four. Full log files and test results will be published online when the study is presented. The summary of the empirical evaluation is presented in table 1. Our base-case is Ubuntu 6.06, a Linux distribution released in 2006 with no countermeasures against code-injection attacks.

6.1 Details for ProPolice

ProPolice is totally focused on protecting the stack and is successful in doing so for direct, stack-based buffer overflows except for structs with a buffer and function pointer. Also indirect, stack-based attacks are prevented because of the re-arranging of character buffers.

On the heap, BSS, and data segment ProPolice does not add any protective countermeasures so direct or indirect, heap/BSS/data-based attacks targeting any of the code pointers and abusing any of the functions will be successful. Indirect, heap/BSS/data-based attacks against longjmp buffers as stack variables or function parameters were not fully stable and thus categorized as partly successful.

6.2 Details for LibsafePlus + TIED

Libsafe's basic protection scheme is wrapping library functions (see list in section 5.3.1. This means that the only stable, successful attack forms were the ones abusing `memcpy()` and RIPE's "homebrew" memcpy equivalent since they are not wrapped.

Direct and indirect, stack/heap/BSS/data-based attacks targeting all code pointers are successful as long as they abuse `memcpy()` or RIPE's "homebrew" memcpy equivalent.

`snprintf()`, `sscanf()`, `strncpy()`, `strncat()`, `sscanf()`, `strcpy()`, `strcat()`, `fscanf()`, and `sprintf()` all were successfully abused a few times and therefore categorized as partly successful. Those partly successful attacks forms were spread across almost all other variables—direct and indirect, stack/BSS/data segment, injected code and return-into-libc and targeting return pointer, longjmp buffers, function pointers, old base pointer and structs.

6.3 Details for CRED

CRED fails to prevent direct and indirect, stack/BSS/data-based overflows toward function pointers, longjmp buffers, and structs for the library functions `sprintf()`, `snprintf()`, `sscanf()`, and `fscanf()`. The attacks against structs are also successful for `memcpy()` and homebrew memcpy equivalent and are the only attacks successful from buffers on the heap. The exception to the above is indirect attacks from the BSS and data segments targeting a longjmp buffer as stack variable. There was some instability in those attacks and therefore they were categorized as partly successful.

6.4 Details for Non-Executable Memory and Stack Protector (Ubuntu 9.10)

Ubuntu 9.10 with non-executable memory and stack protection scored the best in our evaluation. All attack forms that involved the injection of new code in a process' address space failed, due to the policy that a memory page can be either writable, or executable but not both. Also, any attack against the return address of a function was blocked due to the presence of a canary and the re-ordering of variables done by ProPolice. Strackx et. al [45] have shown cases where an attack against the stack is possible even in the presence of canary-based countermeasures, however we decided not to include such an attack in the current version.

All struct attack forms were successful meaning all locations and all abused functions worked, verifying the limitations of ProPolice. Additionally all direct attacks against function pointers on the heap and the data segment were successful. Indirect attacks against the old base pointer works in general on the heap, BSS, and data segment for `memcpy()`, `strcpy()`, `strncpy()`, `sprintf()`, `snprintf()`, `strcat()`, `strncat()`, `sscanf()`, `fscanf()`, and homebrew memcpy equivalent. But there were some instability for 10 of those combinations.

6.5 A Note on Evaluation of StackShield

The testbed execution for StackShield strangely claims only 1810 impossible attack forms whereas all the others say 2990. We figure this is because of StackShield's transformations and have manually removed the missing 1180 impossible attack forms from StackShield's failed attacks since successful and partly successful attacks are obviously possible. If in fact StackShield's transformation makes 1180 extra attack forms possible, that means an increased attack surface and not enhanced protection.

6.6 Potential Shortcomings

6.6.1 Synthesized vs Real-World Code Testbeds

RIPE is a synthesized testbed, deliberately vulnerable, and a program with the sole purpose of conducting attacks against itself and recording their success or failure. Compared to real-world code testbeds, RIPE offers no evaluation of scalability, complexity, or performance. We see merits in both approaches—the main one for synthesized testbeds being the possibility to enumerate and combine attack forms to provide good coverage.

Setup	**Overall effectiveness**	Successful attacks	Partly successful attacks	Failed attacks
Ubuntu 6.06 (no protection)	**0%**	838 (99%)	12 (1%)	0 (0%)
Libsafe (lib wrapper)	**7%**	777 (91%)	16 (2%)	57 (7%)
LibsafePlus (lib wrapper)	**19%**	669 (79%)	16 (2%)	165 (19%)
StackShield (copy & check)	**36%**	533 (63%)	7 (1%)	310 (36%)
ProPolice (canary-based)	**40%**	501 (59%)	9 (1%)	340 (40%)
LibsafePlus+TIED (lib wrapper)	**77%**	170 (20%)	23 (3%)	657 (77%)
CRED (boundary checking)	**79%**	172 (20%)	4 (0.5%)	674 (79%)
Non-exec + stack prot (Ubuntu 9.10)	**89%**	80 (9%)	10 (1%)	760 (89%)

Table 1: Summary of empirical evaluation results using RIPE's 850 buffer overflow attack forms. Overall effectiveness is the percentage of attack forms prevented. Successful attacks give repeatable arbitrary code execution. Partly successful attacks are sometimes successful, sometimes not and in general less stable. Failed attacks are repeatably prevented.

6.6.2 *False Negatives and Result Manipulation*

The kind of evaluation RIPE provides is susceptible to both false negatives and manipulation. An evaluated tool can prevent RIPE's implementation of a given attack form but still allow for exploitations of such attack forms in general. RIPE only provides one vulnerability and matching payload for each of the 850 attack forms, whereas in theory there are infinite variations of both. Such a case could be interpreted as a false negative. Therefore, evaluation results should be interpreted as an upper bound on the preventive effectiveness for the RIPE attack forms—there might be further successful attack forms among the 850.

Further more, researchers could inspect or observe the specifics of how RIPE implements certain attack forms and adjust their countermeasures to prevent exactly those attacks. While this could be based on bad intentions and effectively be result manipulation, it doesn't have to be. Such RIPE-specific prevention might evolve over time when fine tuning to give good evaluation results. Therefore, care has to be taken when comparing RIPE evaluation outcomes between countermeasures. We believe that it is in every researcher's own interest to use RIPE to evaluate fairly and since RIPE is free software any necessary testbed augments can be implemented and published.

7. RELATED WORK

Pincus and Baker present an overview of recent advances in exploitation of buffer overflows [32]. Their main conclusion is that often heard assumptions about buffer overflows are not true—buffer overflows do not all inject code, do not all target the return address, and do not all abuse buffers on the stack. The article briefly discusses (1) injection of attack parameters instead of attack code, (2) attacks targeting function pointers, data pointers, exception handlers, and pointers to virtual function tables in C++, and (3) heap-based overflows.

Michael Zhivich *et al* published "Dynamic Buffer Overflow Detection" in 2005 [54]. They use a collection of 55 small, synthesized C programs that contain buffer overflows to evaluate. They have several "dimensions" in their testbed. They differentiate between *discrete* overflows of up to 8 bytes of memory, and *continuous* overflows resulting from multiple consecutive writes. They have several buffer types—`char`, `int`, `float`, `func *`, and `char *` and they are spread in the same four memory locations as we have; stack, heap, BSS, and data segment. They have buffers in struct, array, union, and array of structs. Lib functions they abuse are `(f)gets`, `(fs)scanf`, `fread`, `fwrite`, `sprintf`, `str(n)cpy`, `str(n)cat`, and `memcpy`. An attack is judged as prevented if it's detected or if a segmentation fault occurs. The top performing tools in their study are Insure++, CCured and CRED. They also evaluate the tools against approximately 100 line models of fourteen historic vulnerabilities in bind, sendmail, and wu-ftpd. CCured, CRED, and TinyCC came out on top, detecting about 90% of the overflows. Unfortunately their testbed is not available which means their study cannot be repeated and their test cases cannot be used for future evaluations. Also they do not try all possible attack combinations nor publish exactly which buffer overflows worked and which didn't. In contrast, RIPE is meant to be a publicly available evaluator which researchers can use to report and compare the coverage of their security mechanisms against a large but well-defined set of real-world attacks.

8. FUTURE WORK

As mentioned in previous sections, RIPE is a process that attacks itself and then checks the success or failure of the launched attacks. Due to the fact that the attack code is part of the vulnerable process, any countermeasures relying on the secrecy of memory locations are defeated since RIPE has access to the addresses of both the overflowing buffer and the target. RIPE could be extended with a *save/load offsets* feature allowing offsets from one execution to be used in a subsequent run. This would allow the evaluation of countermeasures that rely on memory randomization such as ASLR or DieHard [6] and DieHarder [31]. Heap spraying and information leakage attacks could also be added to "assist" an attacker in de-randomizing certain countermeasures. We are also currently considering the addition of non-control data attacks [12] which would allow for evaluation of countermeasures such as *data space randomization*[7] and ValueGuard [48].

9. CONCLUSIONS

Even though hundreds of papers have been published on the problem of buffer overflows and code injection attacks, modern software still is plagued by improper checking of user input attesting to the fact that this is still an open research problem. In this paper we presented RIPE, a Runtime Intrusion Prevention Evaluator which executes a total of 850

buffer-overflow attacks against popular defense mechanisms. The main purpose of RIPE is to provide a freely available testbed which developers of defense mechanisms can use to quantify the security coverage of their proposals and compare themselves against previous work using a well-defined and real-world set of attacks.

Acknowledgements:.

This research is partially funded by the national computer graduate school in computer science (CUGS) commissioned by the Swedish government, the board of education and the Interuniversity Attraction Poles Programme Belgian State, Belgian Science Policy, the IBBT and the Research Fund K.U.Leuven.

Initial parts of the testbed extensions were built by Pontus Viking as part of his Master's Thesis [51].

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

10. AVAILABILITY

RIPE is free software released under the MIT licence and available on GitHub: `https://github.com/johnwilander/RIPE`

11. REFERENCES

[1] Akritidis, P., Markatos, E., Polychronakis, M., and Anagnostakis, K. Stride: Polymorphic sled detection through instruction sequence analysis. In *Security and Privacy in the Age of Ubiquitous Computing*, R. Sasaki, S. Qing, E. Okamoto, and H. Yoshiura, Eds., vol. 181 of *IFIP Advances in Information and Communication Technology*. Springer Boston, 2005, pp. 375–391.

[2] Avijit, K., Gupta, P., and Gupta, D. Tied, libsafeplus: Tools for runtime buffer overflow protection. In *Proceedings of The 13th USENIX Security Symposium* (San Diego, USA, August 2004), pp. 45–56.

[3] Avijit, K., Gupta, P., and Gupta, D. Binary rewriting and call interception for efficient runtime protection against buffer overflows: Research articles. *Softw. Pract. Exper. 36* (July 2006), 971–998.

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

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

[6] Berger, E. D., and Zorn, B. G. Diehard: probabilistic memory safety for unsafe languages. In *Proceedings of the 2006 conference on Programming language design and implementation* (Ottawa, ON, 2006), ACM Press, pp. 158–168.

[7] Bhatkar, S., and Sekar, R. Data space randomization. In *Proceedings of the 5th international conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA '08)* (July 2008).

[8] Blazakis, D. Interpreter Exploitation: Pointer Inference and JIT Spraying. In *BlackHat DC* (2010).

[9] Bray, B. Compiler security checks in depth. `http://msdn.microsoft.com/en-us/library/aa290051(v=vs.71).aspx`, February 2002.

[10] Bulba, and Kil3r. Bypassing StackGuard and StackShield. Phrack Magazine Volume 10, Issue 56 `http://www.phrack.org/phrack/56/p56-0x05`, May 2000.

[11] Castro, M. Securing software by enforcing data-flow integrity. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation* (2006), pp. 147–160.

[12] Chen, S., Xu, J., Sezer, E. C., Gauriar, P., and Iyer, R. K. Non-control-data attacks are realistic threats. In *14th USENIX Security Symposium* (2005).

[13] cker Chiueh, T., and Hsu, F.-H. 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).

[14] Clause, J., Li, W., and Orso, R. Dytan: A generic dynamic taint analysis framework. In *Proceedings of the International Symposium on Software Testing and Analysis* (2007), pp. 196–206.

[15] Costa, M., Crowcroft, J., Castro, M., and Rowstron, A. Can we contain internet worms? In *Proceedings of Third Workshop on Hot Topics in Networks, HotNets-III* (San Diego, CA USA, November 2004).

[16] Costa, M., Crowcroft, J., Castro, M., Rowstron, A., Zhou, L., Zhang, L., and Barham, P. Vigilante: End-to-end containment of internet worms. In *Proceedings of the Symposium on Systems and Operating Systems Principles (SOSP)* (2005), pp. 133–147.

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

[18] Cowan, C., Pu, C., Maier, D., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P., Zhang, Q., and Hinton, H. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Conference* (San Antonio, Texas, January 1998), pp. 63–78.

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

[20] Durden, T. Bypassing pax aslr protection. `http://www.phrack.com/issues.html?issue=59&id=9`, July 2002.

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

[22] Gajdos, P., and Kornacker, C. Cve-2009-4035 xpdf: buffer overflow in fofitype1. `https://bugzilla.redhat.com/show_bug.cgi?id=541614`, December 2009.

[23] grsecurity. Pax. `http://pax.grsecurity.net/`.

[24] Hassell, R., and Permeh, R. Microsoft internet information services remote buffer overflow. http://www.eeye.com/Resources/Security-Center/Research/Security-Advisories/AD20010618, 6 2001.

[25] Howard, M. Evils of strncat and strncpy - answers. http://blogs.msdn.com/b/michael_howard/archive/2004/12/10/279639.aspx, December 2004.

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

[27] Kc, G. S., Keromytis, A. D., and Prevelakis, V. Countering code-injection attacks with instruction-set randomization. In *Proceedings of The 10th ACM Conference on Computer and Communications Security* (Washington D.C., USA, 2003), pp. 272–280.

[28] Moore, D., Paxson, V., Savage, S., Shannon, C., Staniford, S., and Weaver, N. Inside the slammer worm. *IEEE Security and Privacy 1*, 4 (2003), 33–39.

[29] Moore, D., Shannon, C., and claffy, k. Code-red: a case study on the spread and victims of an internet worm. In *2nd ACM Workshop on Internet measurment* (2002).

[30] Nebenzahl, D., and Wool, A. Install-time vaccination of windows executables to defend against stacksmashing attacks. In *Proceedings of The 19th IFIP International Information Security Conference* (2004).

[31] Novark, G., and Berger, E. D. Dieharder: securing the heap. In *Proceedings of the 17th ACM conference on Computer and communications security* (New York, NY, USA, 2010), CCS '10, ACM, pp. 573–584.

[32] Pincus, J., and Baker, B. Beyond stack smashing: Recent advances in exploiting buffer overruns. *IEEE Security and Privacy 2*, 4 (2004), 20–27.

[33] Project, T. U. Ubuntu 9.10 karmic. http://releases.ubuntu.com/karmic/.

[34] Project, T. U. Ubuntu security feature matrix. https://wiki.ubuntu.com/Security/Features.

[35] Qin, F., Wang, C., Li, Z., seop Kim, H., Zhou, Y., and Wu, Y. Lift: A low-overhead practical information flow tracking system for detecting security attacks. *Microarchitecture, IEEE/ACM International Symposium on 0* (2006), 135–148.

[36] Robertson, W., Kruegel, C., Mutz, D., and Valeur, F. Run-time detection of heap-based overflows. In *Proceedings of The 17th Large Installation Systems Administration Conference* (San Diego, USA, October 2003).

[37] Ruwase, O., and Lam, M. S. A practical dynamic buffer overflow detector. In *Proceedings of The 11th Annual Network and Distributed System Security Symposium* (San Diego, USA, February 2004).

[38] Shacham, H. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of CCS 2007* (Oct. 2007), S. De Capitani di Vimercati and P. Syverson, Eds., ACM Press, pp. 552–61.

[39] Shacham, H., Page, M., Pfaff, B., Goh, E.-J., Modadugu, N., and Boneh, D. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM conference on Computer and communications security, CCS '04* (2004).

[40] Sidiroglou, S., and Keromytis, A. D. Countering network worms through automatic patch generation. *Security & Privacy, IEEE 3*, 6 (November–December 2005), 41–49.

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

[42] Smirnov, A., and cker Chiueh, T. Dira: Automatic detection, identification, and repair of control-hijacking attacks. In *Proceedings of The 12th Annual Network and Distributed System Security Symposium* (2005).

[43] Spafford, E. H., and Spafford, E. H. The internet worm program: An analysis. *Computer Communication Review 19* (1988).

[44] SPEC - Standard Performance Evaluation Corporation. http://www.spec.org/.

[45] Strackx, R., Younan, Y., Philippaerts, P., Piessens, F., Lachmund, S., and Walter, T. Breaking the memory secrecy assumption. In *2nd European Workshop on System Security* (2009).

[46] Tuck, N., Calder, B., and Varghese, G. Hardware and binary modification support for code pointer protection from buffer overflow. In *Proceedings of the 37th Intl Symposium on Microarchitecture, MICRO 04* (2004), pp. 209–220.

[47] US-CERT. Vulnerability notes database. http://www.kb.cert.org/vuls.

[48] Van Acker, S., Nikiforakis, N., Philippaerts, P., Younan, Y., and Piessens, F. Valueguard: Protection of native applications against data-only buffer overflows. In *Proceedings of the Sixth International Conference on Information Systems Security (ICISS)* (2010).

[49] van de Ven, A., and Molnar, I. Execshield. http://people.redhat.com/mingo/exec-shield/docs/WHP0006US_Execshield.pdf.

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

[51] Viking, P. Comparison of dynamic buffer overflow protection tools. Master's thesis, Linkopings universitet, February 2006.

[52] Wikipedia. Wikipedia, nx bit. http://en.wikipedia.org/wiki/NX_bit.

[53] Wilander, J., and Kamkar, M. A comparative study of publicly available tools for dynamic buffer overflow prevention. In *Proceedings of the 10th Network & Distributed System Security Symposium* (San Diego, California, February 2003).

[54] Zhivich, M., Leek, T., and Lippmann, R. Dynamic buffer overflow detection. Workshop on the Evaluation of Software Defect Detection Tools, co-located with PLDI 2005 http://ewww.cs.umd.edu/~pugh/BugWorkshop05/, June 2005.