Extended protection against stack smashing attacks without performance loss

Yves Younan[1] Davide Pozza[2] Frank Piessens[1] Wouter Joosen[1]

[1]DistriNet, Dept. of Computer Science
Katholieke Universiteit Leuven
Celestijnenlaan 200a, B3001 Leuven, Belgium

[2]Dip. di Automatica e Informatica
Politecnico di Torino
c.so Duca degli Abruzzi 24, I10129 Torino, Italy

E-mail: [1]{yvesy,frank,wouter}@cs.kuleuven.ac.be [2]Davide.Pozza@polito.it

Abstract

In this paper we present an efficient countermeasure against stack smashing attacks. Our countermeasure does not rely on secret values (such as canaries) and protects against attacks that are not addressed by state-of-the-art countermeasures. Our technique splits the standard stack into multiple stacks. The allocation of data types to one of the stacks is based on the chances that a specific data element is either a target of attacks and/or an attack vector. We have implemented our solution in a C-compiler for Linux. The evaluation shows that the overhead of using our countermeasure is negligible.

1 Introduction

Buffer overflow vulnerabilities are a significant threat to the security of a system. Most of the existing buffer overflow vulnerabilities are located on the stack, and the most common way for attackers to exploit such a buffer overflow is to use it to modify the return address of a function. By making the return address point to code they injected into the program's memory as data, they can force the program to execute any instructions with the privilege level of the program being attacked [2].

According to the NIST's National Vulnerability Database [22], 584 buffer overflow vulnerabilities were reported in 2005, making up 12% of the 4852 vulnerabilities reported that year. In 2004 the amount of reported buffer overflow vulnerabilities was 341 (14% of 2352). This means that while the amount of reported vulnerabilities has almost doubled in the past year buffer overflows still remain an important source of attack. 418 of the 584 buffer overflows reported last year had a high severity rating, this makes up 21% of the 1923 vulnerabilities rated with a high severity level. They also make up 42% of the vulnerabilities which allow an attacker to gain administrator access to a system.

Stack-based buffer overflows have traditionally made up the largest bulk of these buffer overflows, and are the ones most easily exploited by attackers. Many countermeasures have been devised that try to prevent code injection attacks [33]. Several approaches attempt to solve the vulnerabilities entirely [17, 4, 15, 23, 24, 32], however, they generally suffer from a substantial performance impact. Other types of countermeasures have been developed with better performance results that specifically target stack-based buffer overflows. These countermeasures can be divided into four categories. The first category [12, 13] offers protection by using a random value, which must be kept secret from an attacker, if the program leaks this information (e.g. through a 'buffer over-read' or a format string vulnerability) the protection can be bypassed entirely. A second category [29, 10, 5, 31] copies the return address and the saved frame pointer, and compares or replaces them when the function returns. While this protects against the return address being overwritten, it does not protect other information stored on the stack (e.g. pointers) which could be used by an attacker to execute arbitrary code. A third category tries to correct the library functions that are typically the source of an overflow (e.g. *strcpy*) [5], however, this does not protect against buffer overflows which could occur at a different place in the program (e.g. an overflow caused by a loop). A fourth category tries to make attacks harder by modifying the operating system [26, 28, 6, 16, 7] or hardware [31, 20].

In this paper we present a new approach for protecting against stack based buffer overflows by separating the stack into multiple stacks. This separation is done according to the type data stored on the stack. Each stack is protected from writing into the other stack by a guard page[1]. Our countermeasure offers equal or better performance results than the countermeasures in the categories discussed earlier and does not suffer from some of their weaknesses: it does not rely on random numbers and protects pointers as well as

[1]A guard page is page of memory where no permission to read or to write has been set. Any access to such a page will cause the program to terminate.

the return address and frame pointer. In [34] we describe a more global approach to separating control flow data from regular data and in [35] we discuss applying it to the heap to separate the metadata from the regular data.

The paper is structured as follows: section 2 briefly describes the technical details of the stack-based buffer overflow, some representative countermeasures and their weaknesses. Section 3 discusses the design and implementation of our countermeasure. Section 4 evaluates our countermeasure in terms of performance and security. In section 5 we discuss limitations and possible improvements for our approach and describe ongoing work. Section 6 compares our approach to existing countermeasures, while section 7 presents our conclusions.

2 Stack-based buffer overflows

Buffer overflows are the result of an out of bounds write operation on an array. In this section we briefly recap how an attacker could exploit such a buffer overflow on an array that is allocated on the stack.

When an array is declared in C, space is reserved for it and the array is manipulated by means of a pointer to the first byte. At run-time no information about the array size is available and most C-compilers will generate code that will allow a program to copy data beyond the end of an array, overwriting adjacent memory space. If interesting information is stored somewhere in such adjacent memory space, it could be possible for an attacker to overwrite it. On the stack this is usually the case: it stores the addresses to resume execution at after a function call has completed its execution, i.e. the return address.

For example, on the IA32-architecture the stack grows down (i.e. newer function call have their variables stored at lower address than older ones). The stack is divided into stackframes. Each stackframe contains information about the current function: arguments to a function that was called, registers whose values must be stored across function calls, local variables, the saved frame pointer and the return address. An array allocated on the stack will usually be contained in the section of local variables of a stackframe. If a program copies data past the end of this array it will overwrite anything else stored before it and thus will overwrite other data stored on the stack, like the return address.

Several countermeasures were designed against this attack: ranging from bounds checkers to operating system changes. Many of these are discussed in section 6. Here, we discuss two of the mostly used countermeasures that protect against this attack.

StackGuard [12] was designed to be an efficient protection against this type of attack: it protects the return address by placing a randomly generated value (called a canary) between the saved frame pointer and the local variables on the stack. This canary would be generated at program start up and would be stored in a global variable. When a function is called, the countermeasure would put a copy of the canary onto the stack after the saved frame pointer. Before the function returns, the canary stored on the stack will be compared to the global variable, if they differ, the program will be terminated. If an attacker would want to overwrite the return address, he would have to know the canary, so he could write past it. A significant problem with this approach is the fact that the program can not leak the canary, if it did, the attacker could just write the correct value back on the stack and the protection would be bypassed.

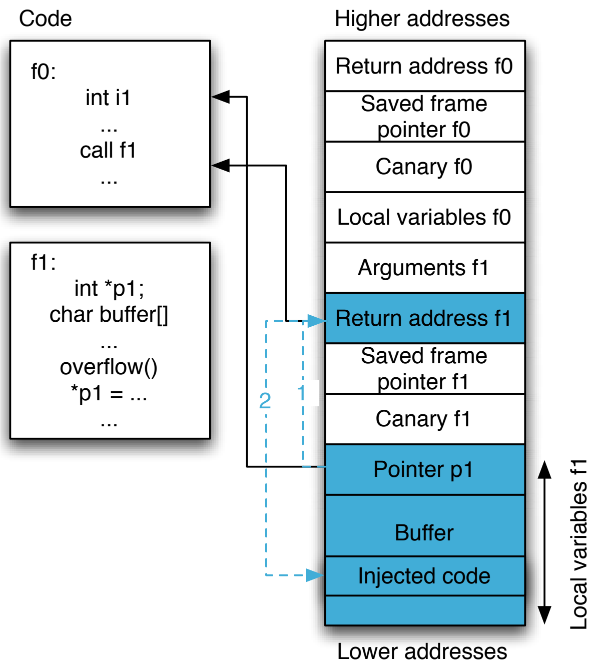

Figure 1. Indirect pointer overwriting attack

Figure 1 depicts the stack layout of a program protected with StackGuard and illustrates an attack called indirect pointer overwriting [8]. This attack consists of exploiting a local buffer to overwrite a pointer p1 stored in the same stackframe and to make the pointer refer to the return address. When the pointer is later dereferenced for writing, it will overwrite the return address rather than the value it was originally pointing to. If attackers can control the value that the program would write via the pointer, they can modify the return address to point to their injected code.

ProPolice [13] attempts to protect against this type of attack by reorganizing the local variables stored in each stackframe: all arrays are stored before all other local variables in each stackframe. This prevents an attacker from using an overflow to overwrite a pointer and using an indirect pointer overwrite to bypass the protection.

However, as mentioned earlier, this type of protection has some limitations: if a program leaks the canary (e.g. through a format string vulnerability or a 'buffer overread'), the protection can be bypassed completely. Another point of attack would be to use a buffer overflow to overwrite an array of pointers in a program or to use a structure that contains a buffer but no pointers to overwrite another structure that does contain such a pointer. It will also not protect against memory that is allocated with the *alloca*[2] call, if an overflow occurs in memory allocated using this call, it could be used to perform an indirect pointer overwrite.

In the next section we discuss our approach which aims to better protect against these type of attacks, while still preserving or improving the performance of the previously described countermeasures.

3 The multiple stacks countermeasure to protect against buffer overflow vulnerabilities

This section describes the approach of the multiple stacks countermeasure by describing the basic concepts behind its design, as well as how it was implemented.

3.1 Approach

The stack stores several kinds of data: some data is related to control flow, such as stored registers, but it also stores regular data like the local variables of a function. However, this regular type of data can sometimes also be used by an attacker to inject code if it is modified (e.g. pointers could allow indirect pointer overwriting). Other data could be used to perform an attack if it is misused.

In this section we describe an approach which separates the stack into multiple stacks based on two criteria: how valuable data is to an attacker when it is a target for attack and the risk of the data being used as an attack vector (i.e. misused to perform an attack). These properties are not mutually exclusive: some data could be both a target and a vector. So, we must evaluate all possible data types and place them in categories according the risk of being an attack vector and the effective value.

We can assign data a ranking based on its risk of being an attack vector and the value it has as a target. Data can have a low, medium, or high ranking for both properties (e.g. the return address has high target value because attackers generally want to overwrite it, and a low vector value because it can't be attacked directly). Based on these rankings we can divide the data into different categories. This is illustrated in Table 1, where we use six categories.

[2]*alloca* is used to dynamically allocate space on the stack, it behaves in much the same way as *malloc* call, except that the memory it allocates will be released when the function returns.

Table 1. Attack vector versus attack target categories

Vector/Target	Low	Medium	High
Low	cat. 3	cat. 2	cat. 1
Medium	cat. 5	cat. 3	cat. 2
High	cat. 5	cat. 4	cat. 6

In principle, one could always argue for other categories or combinations. However, we decided on limiting these categories to six based on how we perceive the combined risk/value resulting from the combination of attack vector risk and attack target value. We believe that the presented set of six categories is a strong trade-off. Our main objective is to show that a multiple stacks countermeasure (based on several categories) can be supported efficiently.

Category one contains highly valuable data and there is only a low risk of it being used as an attack vector. This is the main category that we wish to protect from buffer overflows.

Category two represents two cells from the summary table: data which has a low risk of being an attack vector, but a medium target value, and data which has a medium risk of being a vector, but is also a high-value target. We consider both these two types of data to have a comparable combined risk/value.

Category three contains data which has a medium risk of being a vector, but is also only a medium-value target. We have supplemented it with data which has the least importance in our countermeasure: low on vector-risk and low on target-value. Mainly, it does not matter where this type of information is placed since it needs no protection and can't be used to attack. As such, we decided on placing it in a middle category.

Category four contains data which has a high risk of being an attack vector, but which is also a medium-value target. So, there is some need for protection.

Category five contains data which has a high or medium risk of being a vector, but has only low value as a target. It contains both high and medium risk data, because the data needs to be isolated from higher-value targets, but does not need to be protected.

Category six is the hardest data to protect. It is both a high-value target and has a high risk of being used as an attack vector. We place it in a separate category because it needs both extra protection and needs to be protected from.

We can now decide what information to put in each of these categories by assigning them rankings of their target-value and attack vector-risk.

The Return address is the most obvious target for attack: if an attacker can modify it, he can easily execute injected code. However, an attacker does not directly control the return address, so it is an unlikely vector.

- Attack target: High; Attack vector: Low

Other saved registers on the stack, like the saved frame pointer and the caller-save and callee-save registers could be used to attack a program [19]. So, all these are valuable targets, but generally an attacker can not use them to mount an attack.

- Attack target: High; Attack vector: Low

Pointers can contain reference functions or data. If a function pointer is overwritten, an attacker can directly execute inject code. If a data pointer is overwritten, an attack could use indirect pointer overwriting, so these are very likely targets for attacks. However, they can not be used as an attack vector, unless they can modified by an attacker.

- Attack target: High; Attack vector: Low

Integers can sometimes be used to store pointers or indexes to pointer operations, so they can be considered attack targets. They are not attack vectors in the sense that they could directly overwrite other information on the stack.

- Attack target: Medium; Attack vector: Low

Floating types are not valuable targets because they will not generally contain information that could lead to code injection (either directly or indirectly). They are also unlikely attack vectors because they can't be used directly to overwrite adjacent memory locations.

- Attack target: Low; Attack vector: Low

Arrays are assigned different target values and attack vector-risks depending on their type:

Arrays of pointers are valuable targets, because they contain pointers, and as such could be used to perform an indirect pointer overwrite, if modified. However, there is also a chance that an operation on an array of pointers could lead to writing outside the bounds of the array. Thus, there is a risk of it being used as an attack vector as well. However, these type of arrays are not generally used with functions that are prone to buffer overflows (e.g. *strcpy* and related functions), so this risk is not as high as with arrays of characters.

- Attack target: High; Attack vector: Medium

Arrays of characters are the traditional arrays that are most vulnerable to buffer overflows. The risk of them being used as an attack vector is high, especially since they are also often used with unsafe copying functions. They do not contain any information that could indirectly or directly lead to a code injection attack.

- Attack target: Low; Attack vector: High

Other arrays are possible targets because an integer in an array of integers could be used to store a pointer. As with arrays of pointers, they are possible vectors, since an out of bounds write could occur, but they are not generally used with the most dangerous functions.

- Attack target: Medium; Attack vector: Medium

Arrays of structures and unions are discussed separately.

Structures/unions are assigned different target values and attack vector-risks depending on the type of the data they contain:

Structures containing no arrays at any level (structures and unions can contain other structures or unions) are unlikely attack vectors, but possible targets because they possibly contain pointers.

- Attack target: Medium; Attack vector: Low

Structures containing arrays of characters are likely vectors because a buffer overflow could occur in the character array. They are also possible targets because they could contain pointers.

- Attack target: Medium; Attack vector: High

Structures containing other arrays are possible vectors, because overflows could occur. They are also a target because the structure or union could be used to store a pointer.

- Attack target: Medium; Attack vector: Medium

Arrays of structures/unions Arrays of structures are a special case because the structures or unions stored in such an array can contain arrays at some level.

Not containing arrays of characters If the structures or unions inside the array do not contain arrays of characters at any level, we treat them the same as other arrays: possible targets and possible vectors.

- Attack target: Medium; Attack vector: Medium

Containing arrays of characters As previously mentioned for structures or unions containing character arrays: they are a likely target, and a possible vector.

- Attack target: Medium; Attack vector: High

Based on these assignments and Table 1, the different categories will contain the following data:

Category 1 : return address, other saved registers, pointers.

Category 2 : arrays of pointers, structures and unions (no arrays), integers.

Category 3 : floating types, other arrays, structures/unions containing arrays but not arrays of characters at any levels, arrays of structures that do not contain arrays of characters at any level.

Category 4 : structures containing array of characters, arrays of structures containing arrays of characters

Category 5 : arrays of characters

Category 6 would be the hardest to protect, thankfully it is empty in our risk/value evaluation. There is no data on the stack that we consider to have high risk of being an attack vector but is also a high-value target.

As with the different categories, the actual value that we have assigned specific data is based on the value or risk that we perceive it to have. If some data would be assigned a different risk or value, resulting in it being placed in a different category, this would only require minimal modification of our existing countermeasure.

The main principle used to design this countermeasure is to separate information in these different categories from each other by storing them on separate stacks. As such they can no longer be overwritten by information which has been moved to a different stack. Figure 2 depicts the memory layout if we were to map the five categories that contain data onto five different stacks.

It is however fairly simple to modify our design (and our implementation) to support other stack configurations depending on the amount of risk that these data types or categories present (or if the risk of a particular category or data type can be diminished or abolished entirely) in a particular application versus the amount of memory that can be used. An example of this would be to support only two stacks, and to place categories one, two and three on the first stack, while storing categories four and five on the second stack.

3.2 Implementation

The multiple stack countermeasure was implemented in gcc-4.1-20050902 for Linux on the IA32 architecture. Each such stack is stored sequentially after the other and each stack is protected from the previous using a guard page. We start of by allocating the different stacks at a fixed location from one another. This fixed point is the maximum size that the stack can grow to (this must be known at compile time). As long as no information is written to the specific pages that were allocated for the stack, the program will only use virtual address space, rather than physical address space so we can easily map all stacks into memory without wasting any physical memory.

The countermeasure was implemented in the pass of the compiler that converts the GIMPLE representation[3] into RTL[4].

We implement our countermeasure by modifying the way local variables are accessed in a function. When a function is called in a program, the return address will be stored on the stack, however, to access local variables of a function, the current value of the register containing the stack pointer will be copied to the frame pointer register (and the current saved frame pointer will be saved on the stack). This frame pointer will be used as a fixed location to access a function's local variables (all variables will be accessed as an offset to the frame pointer), this mechanism is used because the value of the register containing the stack pointer is constantly changing whenever a variable is pushed or popped from the stack. The compiler will calculate the offset to the frame pointer for local variables at compile time and will use this offset whenever it accesses this variable. When the function returns, the saved frame pointer will be restored into the frame pointer register.

We use this mechanism to efficiently implement our countermeasure: instead of using multiple stack pointers, we modify the offset to the frame pointer that is used to access the variable. We add $(stacknr - 1) * (sizeofstack + pagesize)$ to the offset, which will result in the access of the variable on the correct stack. As such all operations that use this variable will use the correct stack to address it. This also means we don't incur any overhead because the offset will simply be a larger constant value, but the instruction to access it will remain the same.

Because the program is instrumented in this way, the stack pointer will remain unchanged and effectively controls all five stacks. The advantage is that *setjmp* and

[3]GIMPLE is a language- and target-independent tree representation of the program being compiled. The compiler will convert the program into static single assignment form (SSA) at this level.

[4]RTL is the register transfer language, a language-independent, but target-dependent, intermediate representation used by the the compiler to do some optimizations.

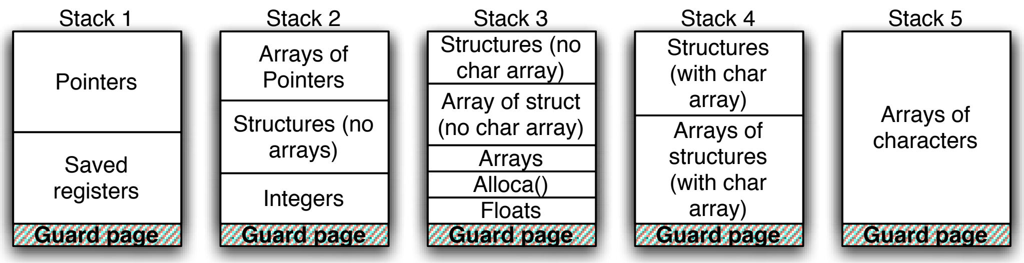

Figure 2. Stack layout for 5 stacks

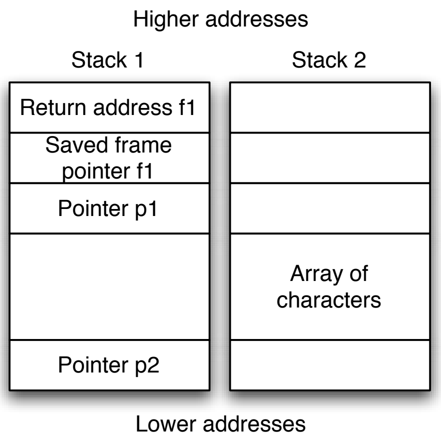

Figure 3. Gaps on the different stacks

longjmp[5] will work unchanged. The drawback of this countermeasure is that it will result in gaps on the remaining stacks, resulting in wasted memory. Figure 3 depicts this for two stacks. We provide a more detailed discussion on the memory overhead in section 4.

A special case which we did not address in the design and the categories above is memory allocated with *alloca*. The information stored in it could be both an attack vector to overwrite other memory and could contain information which could be used to perform a code injection attack (e.g. a function or data pointer). Given this, we chose to modify this call to allocate memory on stack three in the case of five stacks and stack two in the case of two stacks.

4 Evaluation

To test the performance overhead, we ran several benchmarks on programs instrumented with our countermeasure (running with 5 stacks) and without. All tests were run on a single machine (Pentium 4 2.80 Ghz, 512MB RAM, no hyper-threading, running Ubuntu Linux 5.10 with kernel 2.6.12.10). The GCC compiler version 4.1-20050902 was used to compile all benchmarks.

4.1 Performance

This section evaluates our countermeasure in terms of performance overhead. Both macro- and microbenchmarks were performed.

Macrobenchmarks
All programs but one (252.eon is written in C++, while our prototype implementation is only for C) in the SPEC®CPU2000 Integer benchmark [14] were used to perform these benchmarks.

Table 2 contains the amount of code present in a particular program (expressed in lines of code), the runtime in seconds when compiled with the unmodified gcc and the runtime when compiled wih our multistack countermeasure. The results in this table show that the performance overhead of using our countermeasure are negligible for most of these programs. There is a slightly higher overhead of 2-3% for the programs *vortex* and *twolf*. The negative overheads in the table are so low that they can be attributed to normal variations between runs and, as such, these overheads can be considered equal.

Microbenchmarks
Two programs which make extensive use of the stack were run as a microbenchmark. One program which simply calls a function 1 million times. This function performs an addition of two local variables (filled with 'random'[6] values), fills a local array with this random value, and allocates and frees a chunk of random size. The second program performs a recursive Fibonacci calculation of the 42nd Fibonacci number. These programs were each run 100 times both compiled with the unmodified gcc and our multistack countermeasure. Table 2 contains the average runtime in seconds, followed by the standard error for both versions.

[5]The *longjmp* function will jump to the last place in the code where a *setjmp* was executed, resetting the stack pointer (and other registers) to the value they held at the moment *setjmp* was called.

[6]We use a fixed seed for the random function, so the generated values are the same over different runs.

Table 2. Benchmark results

SPEC CPU2000 Integer benchmarks				
Program	LOC	Gcc 4.1 (s)	Multistack (s)	Overhead
164.gzip	8,616	201	201	0%
175.vpr	17,729	213	212	-0.47%
176.gcc	222,182	89.7	89.8	0.11%
181.mcf	2,423	248	249	0.4%
186.crafty	21,150	116	115	-0.86%
197.parser	11,391	257	255	-0.78%
253.perlbmk	85,185	150	151	0.67%
254.gap	71,430	101	101	0%
255.vortex	67,220	169	174	2.96%
256.bzip2	4,649	204	203	-0.49%
300.twolf	20,459	291	297	2.06%
Microbenchmarks				
loop	20	9.166 ± 0.029	9.2 ± 0.015	0.37%
fibonacci	14	3.354 ± 0.004	3.363 ± 0.005	0.27%

These results also confirm that the performance overhead of using our countermeasure is negligible.

4.2 Memory overhead

The maximum memory overhead (which is also the general case) of this countermeasure will be the original stack usage multiplied by the amount of stacks that are used.

Because variables are calculated by simply adding a constant value to the frame pointer, we end up with gaps on all stacks and waste space on all stacks. To reduce the waste, we are planning to implement a version where we calculate the actual location that the variable is on for every stack. This would eliminate gaps in a function entirely. Some gaps would still exist between function calls (because we still only have one stack pointer), but these could be reduced to be equal to the amount of space used on the largest stack. This still allows us to use a single stack pointer, because all other stacks will continue to have gaps, but these gaps will be much smaller than in the current implementation. Since all these calculations can be done at compile time, no extra performance overhead would be incurred.

5 Discussion and ongoing work

Most applications will never increase the default stack size, however, applications that do, may be limited in the size their stack may grow to a predetermined maximum, since the location of the stacks must be set to a fixed location when the program is compiled. However, if the maximum size that the stack could grow to is known beforehand, the locations of the different stacks can easily be moved to accommodate a larger stack. The application would only lose virtual address space when moving the stacks further apart and would not use any extra physical memory until the data is written to these pages. We discuss a possible solution to this problem below.

Our approach is incompatible with most address space layout randomization (ASLR) [28] implementations. This can be mitigated by finding the start of the stack dynamically at program start up, when setting up the extra stacks. This can be done either by recursively following the saved frame pointer values or by modifying the ASLR implementation to store the value in a known location (e.g. the normal stack location) at program start up and subsequently clearing it when the multiple stacks have been set up.

Because not all applications can afford to use five stacks, but would still like more security than simply reducing the amount of stacks to two can offer, we are currently working on extending the countermeasure by a concept which we call selective bounds checking. Selective bounds checking will only bounds check write operations to some types of arrays to prevent them from being overflown. If, for example, we can bounds check write accesses to arrays of pointers, we could determine that the risk of it being used as an attack vector was reduced low enough to place it in the first category. While the bounds checking for direct access is straightforward (we instrument the program to dynamically check if the index is within the bounds of the array), we do static analysis to determine how to instrument indirect accesses to an array. This means that our bounds checker will not find all cases of such accesses, but since we're only interested in reducing the risk of already unlikely attack vectors (like arrays of pointers), this is acceptable. Because most programs do not operate heavily on these unlikely vectors, the performance overhead of adding this type of bounds checking will likely be low.

If this selective bounds checker is applied to reduce the amount of stacks to two, it could be realistic to reserve a register as a stack pointer for this second stack. This would allow us to place this second stack anywhere in memory which would solve both the fixed stack size problem, the incompatibility with ASLR and would eliminate the gaps. However, a possible performance overhead will probably be incurred because this extra register must be modified in much the same way as the original stack pointer.

One vulnerability that is present in existing countermeasures, that we did not address in our countermeasure either is the fact that a structure can contain both a pointer and an array of characters, giving the attacker the possibility to overwrite this pointer using the array of characters. The same is true for memory allocated with alloca (it can be used to store array of characters and pointers). This is an important limitation, however, this type of vulnerability does not occur often in practice, so the limitation is unlikely to significantly undermine the protection.

A non-control data attack [9] that relies on modifying a character array would still work, but is severely limited to only being able to overwrite character arrays.

Our approach also does not *detect* when a buffer overflow has occurred. It is, however, possible to easily and efficiently add such detection as an extension to our implementation by using the technique used by StackGuard and Propolice of placing a random number on the stack and verifying it before returning from the function. This canary would be placed on every stack and compared to the value stored on the first stack before returning. Since the random number is mirrored, we can also use a per function canary, rather than a global one, reducing the risk of an attacker discovering one random number and using it to circumvent the detection in another function. If an attacker does discover the value, the countermeasure will no longer be able to perform detection, but it will not be circumvented, because only the detection and not the security relies on it.

6 Related work

Many countermeasures against code injection attacks exist. In this section, we briefly describe the different approaches that could be applicable for protecting against buffer overflows. The focus is more on the countermeasures which are designed specifically to protect the stack from stack-smashing attacks.

6.1 Protection from attacks on stack-based vulnerabilities

Because the stack-based buffer overflow is a very widespread vulnerability, many countermeasures have been designed to protect against attacks on the stack. In this section we discuss the countermeasures which are most closely related to our countermeasure.

Two related countermeasures, StackGuard [12] and Propolice [13] were both discussed in section 2. They rely on random values that must remain secret to provide protection.

Stack Shield [29] is a countermeasure that attempts to protect against stack smashing attacks by copying the return address to another memory location, before entering the function call and restoring it just before returning from the function. This is an efficient countermeasure and will protect the return address from attack, but will still allow an attacker to use indirect pointer overwriting [8] to bypass the protection.

RAD [10] is similar to Stack Shield, except that it compares the return addresses stored at both locations and will terminate the program if they are different. It solves some compatibility problems of Stack Shied and also better protects the area where the return addresses are copied to. However, it will still only protect return addresses and, thus, could be bypassed using indirect pointer overwriting.

Xu et al. [31] suggest a similar approach to Stack Shield. Their countermeasure splits the stack into a control and a data stack. The control stack stores the return addresses while the data stack contains the rest of the data stored on the stack. Their implementation copies the return address to the the control stack before entering the function call and copies it back from the control stack onto the data stack before returning from the function. The authors provide performance results for the SPEC CPU2000 benchmarks, the performance overheads associated with this approach range from 0.01% for 181.mcf to 23.77% for 255.vortex.

Libverify [5] offers the same kind of protection as Stack Shield: upon entering a function it saves the return address on a return address stack (that it calls a canary stack) and when exiting from a function the saved return address is compared to the actual return address. The main difference with Stack Shield lies in the way that this check is added: Libverify does not require access to the source code of the application, the checks are added by dynamically linking the process with the library at run-time.

Libsafe [5] replaces the string manipulation functions that are prone to misuse with functions that prevent a buffer from being overflown outside its stackframe. This is done by calculating the size of the input string and then making sure that the size of the source string is less than the upper bound of the destination string (the space from the variable's stack location to the saved frame pointer). If it is not smaller, the program will be terminated. Again, as is the case with several other countermeasures, this protection can be bypassed using indirect pointer overwriting.

6.2 Alternative approaches

Other approaches that protect against the more general problem of buffer overflows also protect against stack-based buffer overflows. In this section, we give a brief overview of this work.

6.2.1 Compiler-based countermeasures

Bounds checking [17, 4, 15, 21, 23, 24, 32] is the ideal solution for buffer overflows, however, performing bounds checking in C can have a severe impact on performance or may cause existing object code to become incompatible with bounds checked object code.

Protection of all pointers as provided by PointGuard [11] is an efficient implementation of a countermeasure that will encrypt (using XOR) all pointers stored in memory with a randomly generated key and decrypts the pointer before loading it into a register. To protect the key, it is stored in a register upon generation and is never stored in memory. However, attackers could guess the decryption key if they were able to view several different encrypted pointers. Another attack described in [3] describes how an attacker could bypass PointGuard by partially overwriting a pointer. By only needing a partial overwrite, the randomness can be reduced, making a brute force attack feasible (1 byte: 1 in 256, 2 bytes: 1 in 65536, instead of 1 in 2^{32}).

6.2.2 Operating system-based countermeasures

Non-executable memory [26, 28] tries to prevent code injection attacks by ensuring that the operating system does not allow execution of code that is not stored in the text segment of the program. This type of countermeasure can, however, be bypassed by a return-into-libc attack [30] where an attacker executes existing code (possibly with different parameters).

Randomized instruction sets [6, 16] also try to prevent an attacker from executing injected code by encrypting instructions on a per process basis while they are in memory and decrypting them when they are needed for execution. However, software based implementations of this countermeasure incur large performance costs, while a hardware implementation is not immediately practical. Determined attackers may also be able to guess the encryption key and, as such, be able to inject code [27].

Address randomization [28, 7] is a technique that attempts to provide security by modifying the locations of objects in memory for different runs of a program, however, the randomization is limited in 32-bit systems (usually to 16 bits for the heap) and as a result may be inadequate for a determined attacker [25].

6.2.3 Execution monitoring

In this section we describe two countermeasures that will monitor the execution of a program and will prevent transferring control-flow which could be unsafe.

Program shepherding [18] is a technique that will monitor the execution of a program and will disallow control-flow transfers[7] that are not considered safe. Program shepherding can be used for example to ensure that programs can only jump to entry points of functions or libraries, denying an attacker the possibility of bypassing checks that might be performed before a certain action is taken in a function. Another example of a use for program shepherding is to enforce return instructions to only return to the instruction after the call site. The proposed implementation of this countermeasure is done using a runtime binary interpreter, as a result the performance impact of this countermeasure is significant for some programs, but acceptable for others.

Control-flow integrity [1] determines a program's control flow graph beforehand and ensures that the program adheres to it. It does this by assigning a unique ID to each possible control flow destination of a control flow transfer. Before transferring control flow to such a destination, the ID of the destination is compared to the expected ID, and if they are equal, the program proceeds as normal. Performance overhead may be acceptable for some applications, but may be prohibitive for others.

7 Conclusion

In this paper we described a countermeasure that protects against stack-based buffer overflows which has negligible performance overhead, while solving some of the shortcoming of existing efficient countermeasures. We assign all the different data types stored on the stack a high, medium or low ranking, both for the risk of it being an attack vector and the value it has as a possible target. Using this information we assigned the data on the stack to different categories. Each of these categories was then mapped onto a separate stack. This effectively separated high-value targets from data which has a high risk of being used to launch an attack. A straight mapping of categories resulted in an implementation which has a very low performance overhead and offers better protection than existing countermeasures. However, the memory usage in our implementation is higher than most other countermeasures, and we discuss possible ways to reduce it. One of the important advantages of our approach over existing approaches, is that it does not rely on the secrecy of canaries. Our countermeasure remains secure even if an attacker is able to read arbitrary memory locations.

[7]Such a control flow transfer occurs when e.g. a *call* or *ret* instruction is executed.

References

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity. In *Proc. of the 12th ACM Conf. on Computer and Communications Security*, Alexandria, VA, Nov. 2005.

[2] Aleph One. Smashing the stack for fun and profit. *Phrack*, 49, 1996.

[3] S. Alexander. Defeating compiler-level buffer overflow protection. *;login: The USENIX Magazine*, 30(3), June 2005.

[4] T. M. Austin, S. E. Breach, and G. S. Sohi. Efficient detection of all pointer and array access errors. In *Proc. of the ACM SIGPLAN '94 Conf. on Programming Language Design and Implementation*, Orlando, FL, June 1994.

[5] A. Baratloo, N. Singh, and T. Tsai. Transparent run-time defense against stack smashing attacks. In *USENIX 2000 Annual Technical Conf. Proc.*, San Diego, CA, June 2000.

[6] E. G. Barrantes, D. H. Ackley, S. Forrest, T. S. Palmer, D. Stefanović, and D. D. Zovi. Randomized instruction set emulation to disrupt binary code injection attacks. In *Proc. of the 10th ACM Conf. on Computer and Communications Security*, Washington, DC, Oct. 2003.

[7] S. Bhatkar, D. C. DuVarney, and R. Sekar. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *Proc. of the 12th USENIX Security Symp.*, Washington, DC, Aug. 2003.

[8] Bulba and Kil3r. Bypassing Stackguard and stackshield. *Phrack*, 56, 2000.

[9] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer. Non-control-data attacks are realistic threats. In *Proc. of the 14th USENIX Security Symp.*, Baltimore, MD, Aug. 2005.

[10] T. Chiueh and F.-H. Hsu. RAD: A compile-time solution to buffer overflow attacks. In *Proc. of the 21st Int. Conf. on Distributed Computing Systems*, Phoenix, AZ, Apr. 2001.

[11] C. Cowan, S. Beattie, J. Johansen, and P. Wagle. PointGuard: protecting pointers from buffer overflow vulnerabilities. In *Proc. of the 12th USENIX Security Symp.*, Washington, DC, Aug. 2003.

[12] C. Cowan, C. Pu, D. Maier, H. Hinton, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proc. of the 7th USENIX Security Symp.*, San Antonio, TX, Jan. 1998.

[13] H. Etoh and K. Yoda. Protecting from stack-smashing attacks. Technical report, IBM Research Divison, Tokyo Research Laboratory, June 2000.

[14] J. L. Henning. Spec cpu2000: Measuring cpu performance in the new millennium. *Computer*, 33(7), July 2000.

[15] R. W. M. Jones and P. H. J. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In *Proc. of the 3rd Int. Workshop on Automatic Debugging*, Linköping, Sweden, 1997.

[16] G. S. Kc, A. D. Keromytis, and V. Prevelakis. Countering code-injection attacks with instruction-set randomization. In *Proc. of the 10th ACM Conf. on Computer and Communications Security*, Washington, DC, Oct. 2003.

[17] S. C. Kendall. Bcc: Runtime checking for C programs. In *Proc. of the USENIX Summer 1983 Conf.*, Toronto, ON, July 1983.

[18] V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure execution via program shepherding. In *Proc. of the 11th USENIX Security Symp.*, San Francisco, CA, Aug. 2002.

[19] klog. The frame pointer overwrite. *Phrack*, 55, 1999.

[20] R. B. Lee, D. K. Karig, J. P. McGregor, and Z. Shi. Enlisting hardware architecture to thwart malicious code injection. In *Proc. of the First Int. Conf. on Security in Pervasive Computing*, volume 2802 of *LNCS*, 2003.

[21] K.-S. Lhee and S. J. Chapin. Type-assisted dynamic buffer overflow detection. In *Proc. of the 11th USENIX Security Symp.*, San Francisco, CA, Aug. 2002.

[22] National Institute of Standards and Technology. National vulnerability database statistics. `http://nvd.nist.gov/statistics.cfm`.

[23] Y. Oiwa, T. Sekiguchi, E. Sumii, and A. Yonezawa. Fail-safe ANSI-C compiler: An approach to making C programs secure: Progress report. In *Proc. of Int. Symp. on Software Security 2002*, Tokyo, Japan, Nov. 2002.

[24] O. Ruwase and M. S. Lam. A practical dynamic buffer overflow detector. In *Proc. of the 11th Annual Network and Distributed System Security Symp.*, San Diego, CA, Feb. 2004.

[25] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. On the Effectiveness of Address-Space Randomization. In *Proc. of the 11th ACM Conf. on Computer and communications security*, Washington, DC, Oct. 2004.

[26] Solar Designer. Non-executable stack patch. `http://www.openwall.com`.

[27] N. Sovarel, D. Evans, and N. Paul. Where's the FEEB? the effectiveness of instruction set randomization. In *Proc. of the 14th USENIX Security Symp.*, Baltimore, MD, Aug. 2005.

[28] The PaX Team. Documentation for the PaX project. `http://pageexec.virtualave.net/docs/`.

[29] Vendicator. Stack shield. `http://www.angelfire.com/sk/stackshield`.

[30] R. Wojtczuk. Defeating Solar Designer's Non-executable Stack Patch. Bugtraq mailinglist, 1998.

[31] J. Xu, Z. Kalbarczyk, S. Patel, and K. I. Ravishankar. Architecture support for defending against buffer overflow attacks. In *Second Workshop on Evaluating and Architecting System dependabilitY*, San Jose, CA, Oct. 2002.

[32] W. Xu, D. C. DuVarney, and R. Sekar. An Efficient and Backwards-Compatible Transformation to Ensure Memory Safety of C Programs. In *Proc. of the 12th ACM SIGSOFT Int. Symp. on Foundations of Software Engineering*, Newport Beach, CA, October-November 2004.

[33] Y. Younan, W. Joosen, and F. Piessens. Code injection in C and C++ : A survey of vulnerabilities and countermeasures. Technical Report CW386, Departement Computerwetenschappen, Katholieke Universiteit Leuven, July 2004.

[34] Y. Younan, W. Joosen, and F. Piessens. A methodology for designing countermeasures against current and future code injection attacks. In *Proc. of the Third IEEE Int. Information Assurance Workshop 2005*, College Park, MD, Mar. 2005.

[35] Y. Younan, W. Joosen, and F. Piessens. Efficient protection against heap-based buffer overflows without resorting to magic. In *Proc. of the Int. Conf. on Information and Communication Security*, Raleigh, NC, Dec. 2006.