ValueGuard: Protection of native applications against data-only buffer overflows

Steven Van Acker, Nick Nikiforakis, Pieter Philippaerts, Yves Younan, and Frank Piessens

IBBT-Distrinet
Katholieke Universiteit Leuven
3001 Leuven
Belgium
Steven.VanAcker@student.kuleuven.be
{Nick.Nikiforakis,Pieter.Philippaerts,yvesy,frank}@cs.kuleuven.be

Abstract. Code injection attacks that target the control-data of an application have been prevalent amongst exploit writers for over 20 years. Today however, these attacks are getting increasingly harder for attackers to successfully exploit due to numerous countermeasures that are deployed by modern operating systems. We believe that this fact will drive exploit writers away from classic control-data attacks and towards data-only attacks. In data-only attacks, the attacker changes key data structures that are used by the program's logic and thus forces the control flow into existing parts of the program that would be otherwise unreachable, e.g. overflowing into a boolean variable that states whether the current user is an administrator or not and setting it to "true" thereby gaining access to the administrative functions of the program.
In this paper we present ValueGuard, a canary-based defense mechanism to protect applications against data-only buffer overflow attacks. ValueGuard inserts canary values in front of all variables and verifies their integrity whenever these variables are used. In this way, if a buffer overflow has occurred that changed the contents of a variable, ValueGuard will detect it since the variable's canary will have also been changed. The countermeasure itself can be used either as a testing tool for applications before their final deployment or it can be applied selectively to legacy or high-risk parts of programs that we want to protect at run-time, without incurring extra time-penalties to the rest of the applications.

Keywords: buffer overflows, non-control-data attacks, canary

1 Introduction

The buffer overflow is probably the most widely known programming error. It has been used by attackers for over 20 years to exploit programs that do poor handling of user input. The most known computer worms, Morris Worm [31], Code Red [23] and SQL Slammer [22] all used a buffer overflow in vulnerable software as their primary way of attacking and infecting new hosts. Even though

the attack is well understood and many solutions have been proposed over the years, buffer overflows continue to plague modern and legacy software, which is written in unsafe languages. SANS application security blog currently ranks the "classic buffer overflow" as third in their list of twenty-five most dangerous programming errors [29].

Buffer overflows are commonly associated with an attacker placing code of his choice in a variable of the vulnerable program and then using the overflow itself to overwrite a memory location that is used to dictate the control-flow of the running program. Such memory locations are return-addresses, saved base pointers, function pointers and so on. These attacks are called control-data attacks since they target data that is used to control the application's behavior. Since these attacks are the most prevalent, academics and the programming industry itself has focused most of their efforts in protecting the control-data of an application. Stackguard [12] and DEP [21], two widely used countermeasures in modern operating systems are geared towards protecting control-data attacks. The former protects the return address in each stack-frame from overwrites by placing a canary in-front of it and checking its integrity before the function is allowed to return. The latter tries to stop an attacker by marking the stack and the heap memory pages of the current running process as non-executable. Even if an attacker somehow manages to gain control of the execution-flow of the process, he can no longer execute code that he earlier injected.

Since successful exploitation of control-data attacks is becoming harder by the day, it is reasonable to assume that attackers will change their focus into a new exploiting technique that will give them as much control as the old ones. Data-only, or non-control data, attacks fit this description. In non-control data attacks, the attacker is no longer trying to inject and execute his own code. He identifies the existing portions of a program that are of interest to him (e.g. the functions that are allowed to run by an administrator) and he changes the values of data structures in the program that will enable him to access functionality that he normally couldn't (e.g. change the boolean value of a variable that encodes whether the current user is an administrator). Many of the countermeasures proposed to mitigate classic control-data attacks cannot detect non-control data attacks (including the aforementioned Stackguard and DEP).

In this paper we present ValueGuard, a countermeasure specifically geared towards preventing non-control data attacks. ValueGuard identifies all variables in the source code of a program and protects each one individually by placing a random value, a canary, in-front of it. If an attacker uses a buffer-overflow to change the contents of a variable, he will inevitably overwrite over the canary before writing into the variable itself. ValueGuard checks the integrity of a variable's canary before any expression that uses the value of that variable. If the canary has been changed, it is a sign of a non-control data attack and ValueGuard forces the process to terminate, effectively stopping the attack.

Depending on how critical an application is, ValueGuard can be used either as a testing tool to find vulnerabilities before the actual deployment or as a run-time protection tool which will detect and stop data-only buffer overflows in

time. While testing the effectiveness of our system, we discovered a heap-based buffer overflow vulnerability in the Olden benchmark suite that was previously unreported.

The rest of this paper is structured as follows. In Section 2 we describe the different categories of non-control data that an attacker can misuse followed by an example program vulnerable to a non-control data attack. In Section 3 we present the design of our countermeasure and in Section 4 we give details concerning our specific implementation. In Sections 5 and 6 we evaluate the security of ValueGuard and the performance of our prototype. Related work is discussed in Section 7 and we conclude in Section 8.

2 Data-only or Non-control-data attacks

In this section we present the different data structures that a non-control data attack may target and we give an example of a program vulnerable to such an attack.

2.1 Critical Data Structures

Chen et. al [9] were among the first researchers to point out that non-control data attacks can be as dangerous as control data attacks. In their paper, they experimented with real-world applications and they showed that an attacker trying to conduct a non-control data attack, has a number of critical data structures at his disposal which he can overwrite to compromise a running application. Their study showed that these data structures can be categorized in four different types:

Configuration data
: Data stored in a process's memory that was read from e.g. a configuration file. The process expects this data to be specified by the system administrator. If an attacker can overwrite such data, the process's behavior can change in ways the system administrator could not foresee.

User identity data
: Data that identifies a user after e.g. a login, is typically used to enforce access to resources. If this data is altered, an attacker could impersonate another user and get unauthorized access to the user's resources.

User input string
: User input validation ensures that user input conforms to the format a program expects when handling it. If an attacker manages to change the input string after it has been validated, then the program will consider the input safe while it is not.

Decision making data
: Overwriting data used to make decisions can obviously have disastrous consequences. Our example attack in Section 2.2 targets decision making data.

It is clear that at least a subset of these types of data structures is present in any useful real-world application. While their exploitation is not as straightforward as in control-data attacks and the attacker needs to be able to at least partially understand the semantics of a program, Chen et. al showed that it can be done. We argue that today, Chen's observation that "non-control data attacks are realistic threats" is as relevant as ever. A program which would otherwise be not exploitable because of the deployed countermeasures may be vulnerable to a non-control data attack.

2.2 Non-control data attack

```
1  int main(int argc, char **argv) {
2      char pass[40];
3      int authenticated = 0;
4      char buffer[30];
5      char *p;
6
7      readPassFile(PASSFILE, pass, sizeof(pass));
8
9      printf("Enter password: ");
10     fgets(buffer, sizeof(pass), stdin);
11
12     if(!strcmp(buffer, pass)) { authenticated = 1; }
13
14     if(authenticated) {
15         printf("Yes!\n");
16         execl("/bin/sh", "sh", NULL);
17     }
18
19     return 0;
20 }
```

Fig. 1. Example code of a data-only vulnerability.

Consider the program listed in Figure 1. The purpose of the program is to authenticate a user. If the user supplies the correct password, he is given a shell else the program exits. The `main()` function contains a call to the `fgets()` function to read a line of text from the user, on line 10. While `fgets()` is considered a safe function since its second argument states the maximum number of characters to be read, the programmer misused the argument and instead of providing `fgets()` with the size of `buffer`, it provided the size of `pass`. So `fgets()` will read up to 40 characters, which is 10 more than the size of `buffer`. This is a typical example of a buffer overflow.

When this program is compiled with stack smashing protection in place ([12, 14, 30]), this vulnerability can not be exploited to initiate a control-data attack. However, when the `buffer` variable is overflowed the `authenticated` variable is overwritten since the two variables are adjacent on the stack. This variable is normally set by the program when the authentication was successful. An attacker could overflow this value and set it to a non-zero value. The program will think that authentication succeeded, even though it didn't, and it will execute the `/bin/sh` shell.

3 ValueGuard Design

The design of ValueGuard is based on the concepts introduced by StackGuard and extends them to cover all variables instead of only protecting the return address. Naturally, this will result in a higher performance overhead, but this way one can reliably detect bugs or attacks that corrupt only part of the stack or heap.

During the compilation of a program, the ValueGuard framework rewrites the source code of the application and encapsulates all variables into *protection structures*. A protection structure is implemented as C `struct` that consists of two items: the original variable and a canary value. When a variable is allocated, either on the stack or heap, the canary value is initialized to a random value that changes on every run of the application. The application is further modified to detect when every variable is used, and additional canary checks are inserted accordingly.

Pointer Support An important requirement is to detect changes to variables that are used indirectly through pointers. Figure 2 shows an application that complicates the verification of the canary of the 'important' variable, because it is accessed through a pointer. Figure 3 shows the stack contents for the program during a normal run. If an attacker manages to abuse the call to *strcpy* to overwrite the value (and canary) of the 'important' variable, this will not be detected. When the pointer variable 'p' is dereferenced, a check will be executed that verifies the canary of 'p' itself, which was unchanged in the attack - Fig. 4.

The source of the problem is that a variable is used through a pointer, without first checking the integrity of the canary in front of that variable. The detection mechanism of ValueGuard solves this by adding checks for each pointer dereference that looks up and verifies the canary of the dereferenced variable. This lookup is necessary because the pointers may point to objects that are un-predictable (or un-decidable) at compile-time.

ValueGuard uses a memory map to store information about all the registered objects in memory space. When objects are created, they are registered in this memory map. On each pointer dereference, the corresponding memory object can be looked up and the associated canary can be verified.

```
1  int main(int argc, char **argv) {
2      int important = 123;
3      char buffer[80];
4      int *p = &important;
5
6      strcpy(buffer, argv[1]);
7      printf("%d\n", *p);
8
9      return 0;
10 }
```

Fig. 2. An example that shows how pointer de-references can complicate canary verification

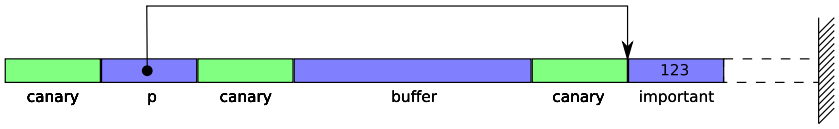

Fig. 3. Stack during a normal run of the program

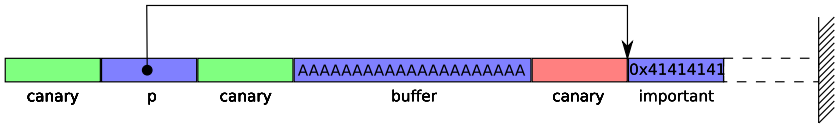

Fig. 4. Stack after malicious `strcpy()`

Compatibility Not all code in a process can be assumed to have been instrumented. Code in shared libraries will not be aware of the use of canaries. ValueGuard, unlike similar countermeasures [16, 4, 15], does not change the representation of pointers or the calling conventions of functions and thus remains fully compatible with existing code. This also implies that ValueGuard supports being used selectively. Developers can choose to protect only (potentially crucial) parts of an application with ValueGuard, thereby limiting the total overhead.

ValueGuard does not change the layout of structures that are defined by the programmer, because many programmers rely on the exact layout of objects in memory. As a result of this, ValueGuard cannot insert canaries in-between the different fields of a structure. Hence, buffer overflows that corrupt data inside a single structure are not detected. This is a limitation of most existing defense mechanisms.

4 Prototype implementation

4.1 Canaries

Our implementation makes use of the CIL framework ([24]) to transform C-code. During this transformation, modifications are made to the code through a custom CIL plugin.

Canaries are implemented as integers that are encapsulated in a struct, together with the variable they protect. An array of random canary values is initialized at program start. Each canary is initialized with a value from that table and the indices into the canary table are determined at compile-time. During the code transformation, canary verification calls are inserted in front of statements that use protected variables. If a verification fails, the program is forced to terminate with a segmentation fault.

Several optimizations are introduced to reduce overhead. First, safe variables are grouped together and protected by a single canary. Safe variables are those variables whose addresses are never used. Arrays and variables used with the address operator (`&`) are therefore unsafe. Second, multiple canary verification calls can be made in sequence, all verifying the same canaries. These are obviously grouped together in a single call. Third, verification calls preceded by safe statements are shifted upwards so that they are grouped. Safe statements are the kind of statements that do not threaten the integrity of any canaries, for example assigning the result of a calculation to a variable. Last, safe functions are not instrumented with extra code. A safe function is one that only uses safe statements and local variables.

4.2 Memory map

The memory map stores the start addresses of memory objects. For every block of 2^k bytes in the memory space, there is an entry in the memory map that holds the start address of the memory object it holds. Memory objects must be aligned to and be a multiple of 2^k bytes. To handle registration of memory objects on the heap, the memory allocator functions `malloc`, `calloc`, `realloc` and `free` are overridden with a wrapper. The wrapper functions allocate extra memory for a canary value, initialize the canary and register the memory object with the memory map. An extra verification call is inserted in front of every pointer dereference. The pointer is looked up in the memory map. If it points to a protected object, the associated canary is verified. Just as for failed regular canary verification, the process is terminated if the canary can not be verified.

5 Security evaluation

In this section we evaluate the security provided by ValueGuard and we present cases that show that ValueGuard can detect attacks in real-world scenarios.

5.1 Effectiveness

ValueGuard's effectiveness in detecting data-only buffer overflows lies in the accurate detection of a modified canary. As explained in Section 4 the canary of each variable is a random integer number chosen at the runtime of the protected program. In order for an attacker to evade detection while using a buffer overflow to conduct a non-control data attack, he must be able to restore the canary to its original contents. This can be done by a) brute-forcing the canary or b) finding out the value of the canary through a memory leakage attack.

Brute-forcing : When ValueGuard detects a modified canary, it terminates the running process. This means that for a canary of 4 bytes (standard integer size) the attacker must make, for the worst-case scenario, 2^{32} attempts before finding out the correct value. Accordingly, ValueGuard will terminate the process $2^{32}-1$ times before the attacker succeeding. We believe that a system's administrator is likely to notice that an attack is taking place well before the exhaustion of 4 billion attempts.

Memory Leakage : Strackx et. al [33] have shown how certain programming errors can reveal to the attacker parts of memory that he can use to de-randomize countermeasures that rely on secret data. While this attack is possible we believe that its exploitation in the case of ValueGuard is not probable since the attacker must find the canary for the specific variables that he can overflow and not just any secret canary. That is because ValueGuard uses multiple canaries and thus the compromise of one canary doesn't necessarily lead to a compromise of the whole countermeasure.

In total, practice shows that the randomness provided by 32 bits of data is enough to ensure security. While some may argue that bounds-checkers provide better security guarantees since their detection is not related with random values, we would like to point out that in our case, ValueGuard will detect overflows occurred while in third-party code (such as libraries) while bounds-checkers will not. This is because bounds-checkers can detect overflows only in code that they have instrumented and thus can't protect variables when third-party code (such as external libraries) accesses them. ValueGuard on the other hand, will be able to detect that an overflow occurred since the variable's canary will have been changed regardless of where the overflow happened.

5.2 Real world test

The defense mechanism was tested in the real world at the Hackito Ergo Sum 2010 conference in Paris. During a 3 day period, a hacker wargame was hosted which contained a program compiled with the described defense mechanism. The program contained a data-only vulnerability that could lead to a root-compromise. After the conference, the wargame was moved to the OverTheWire

([26]) wargame network where it still resides. Despite the numerous attempts, the program was not exploited.

The techniques used in the attacks were closely observed. We found out that most attackers un-successfully tried to circumvent the countermeasure by the means of guessing the values generated by the random number generator.

5.3 Heap overflow in em3d

During the benchmarks, a heap overflow was detected by the defense mechanism in the em3d test of the Olden benchmarks. The overflow occurs in the `initialize_graph` function in `make_graph.c`: The assignment `retval->e_nodes[i] = local_node_r;` is executed for `i = 1 to NumNodes` where `NumNodes` is a command-line parameter, while the `e_nodes` array in `retval` only has room for a fixed amount of values (determined by the `PROCS` constant in `em3d.h`)

The discovery of this heap overflow in a commonly used benchmark suite like Olden, is further validation that ValueGuard can detect and stop non-control data attacks. To our knowledge, no other defense mechanism has detected this overflow before.

6 Performance evaluation

The extra calls to verify the integrity of canaries, have an impact on runtime and memory usage. To measure this effect, two benchmark suites were run: Olden and SPEC CPU2000. All benchmarks were run on Dell GX755 machines with each an Intel Core 2 Duo CPU (E6850) running at 3.00GHz and 4GB of memory, running Ubuntu GNU/Linux 8.04 LTS with kernel 2.6.24-27-server.

Each benchmark was compiled with 5 “compilers”:

gcc The GNU C compiler, version 4.2.4 (Ubuntu 4.2.4-1ubuntu4)

cilly Transformation with the CIL driver w/o any modules, compilation with gcc.

vg_baseline
Using the ValueGuard plugin, but with all flags turned off. This is basically the same as the cilly compiler without any modules.

vg_stackdatabss
Using the ValueGuard plugin, but with the memory map disabled. Only the stack, data and BSS variables are protected.

vg_all
Using the full defense mechanism.

Some tests from the two benchmark suits were ommited either because they were not compatible with the CIL transformation framework or because ValueGuard detected an overflow (see Section 5.3) and terminated the running test. The runtime and memory usage results for Olden and SPEC CPU2000 can be found in Figures 5, 6, 7 and 8.

	gcc	cilly	vg_baseline	vg_stackdatabss	vg_all
bh	101.12 (±7.03)	100.24 (±4.97)	100.64 (±5.74)	153.36 (±9.03)	282.00 (±13.10)
bisort	23.63 (±0.21)	23.69 (±0.20)	23.67 (±0.21)	28.40 (±0.27)	43.04 (±0.47)
health	3.40 (±0.10)	3.40 (±0.09)	3.41 (±0.08)	3.80 (±0.09)	10.84 (±0.13)
mst	4.69 (±0.08)	4.70 (±0.07)	4.70 (±0.08)	7.14 (±0.09)	10.52 (±0.16)
perimeter	1.12 (±0.03)	1.06 (±0.03)	1.05 (±0.03)	1.31 (±0.03)	2.15 (±0.04)
treeadd	21.94 (±0.23)	21.84 (±0.07)	24.42 (±0.46)	24.30 (±0.44)	37.68 (±0.49)
tsp	22.76 (±0.10)	22.76 (±0.12)	22.82 (±0.19)	25.27 (±0.11)	29.10 (±0.17)
Average	25.52	25.39	25.82	34.80	59.33

Fig. 5. Olden benchmarks: runtime results in seconds. Lower is better. The values in between brackets are the standard deviation.

	gcc	cilly	vg_baseline	vg_stackdatabss	vg_all
bh	70.47	70.49	70.49	70.50	128.79
bisort	128.43	128.44	128.45	128.45	640.48
health	147.25	147.27	147.27	147.27	657.47
mst	312.72	312.74	312.74	312.74	391.51
perimeter	171.09	171.10	171.10	171.10	533.80
treeadd	256.43	256.44	256.44	256.44	1280.51
tsp	320.49	320.51	320.51	320.51	960.58
Average	200.98	201.00	201.00	201.00	656.16

Fig. 6. Olden benchmarks: memory usage in MiB. Lower is better.

	gcc	cilly	vg_baseline	vg_stackdatabss	vg_all
164.gzip	99.68 (±0.14)	98.07 (±0.17)	98.71 (±0.08)	183.77 (±0.64)	188.83 (±0.28)
181.mcf	55.77 (±0.26)	55.55 (±0.39)	55.43 (±0.05)	91.87 (±0.66)	171.11 (±0.48)
196.parser	1046.60 (±37.57)	1014.33 (±7.62)	1003.64 (±12.64)	1082.86 (±1.37)	1307.27 (±14.65)
254.gap	48.01 (±0.18)	50.62 (±0.21)	49.62 (±0.09)	145.44 (±0.27)	410.74 (±1.01)
256.bzip2	78.45 (±0.19)	78.70 (±0.20)	79.09 (±0.30)	176.36 (±1.43)	277.06 (±0.41)
300.twolf	113.43 (±0.39)	116.52 (±0.12)	116.90 (±0.17)	394.75 (±10.38)	539.25 (±10.25)
177.mesa	89.36 (±0.76)	80.22 (±0.26)	81.41 (±0.95)	120.44 (±0.31)	472.93 (±6.97)
179.art	77.68 (±0.40)	78.06 (±0.26)	77.73 (±0.72)	139.63 (±0.54)	294.25 (±3.31)
183.equake	56.97 (±0.13)	55.95 (±0.02)	56.15 (±0.03)	91.35 (±0.01)	483.01 (±4.62)
Average	185.11	180.89	179.85	269.61	460.50

Fig. 7. SPEC CPU2000 benchmarks: runtime results in seconds. Lower is better. The values in between brackets are the standard deviation.

When not using the memory map, the results show 25% and 100% average runtime overhead for the Olden and SPEC CPU2000 benchmarks. The memory overhead is negligible in this case (0% for Olden, 1% for SPEC CPU2000).

Using the memory map comes at a cost. For the Olden benchmarks, the overhead increases to 114% and for SPEC CPU2000 it increases to 351% overhead. Likewise, the memory usage due the memory map increases as well: 238% over-

	gcc	cilly	vg_baseline	vg_stackdatabss	vg_all
164.gzip	2712.27	2712.51	2712.52	2713.04	3391.16
181.mcf	232.13	232.19	232.19	232.23	357.70
197.parser	76.62	76.68	76.66	77.07	100.23
254.gap	579.10	579.13	579.13	580.61	728.34
256.bzip2	1666.35	1666.50	1666.52	1666.84	2083.29
300.twolf	12.05	12.09	12.11	12.96	36.09
177.mesa	27.09	27.13	27.14	27.50	46.45
179.art	22.84	22.96	22.96	23.05	39.26
183.equake	125.84	125.90	125.90	125.98	390.69
Average	606.03	606.12	606.13	606.59	797.02

Fig. 8. SPEC CPU2000 benchmarks: memory usage results in MiB. Lower is better.

head for Olden and 79% for SPEC CPU2000. We believe that developers can use the full version of ValueGuard while testing their applications before deployment and the basic version (ValueGuard without the memory map) after deployment. This will allow them to detect and correct as many programming errors as possible while at development phase where the performance of applications doesn't matter. For deployed applications, the basic mode of ValueGuard can be chosen to protect the running applications with an acceptable performance cost.

7 Related work

Many approaches exist that try and protect against buffer overflow attacks. In this section we will briefly discuss the most important types of countermeasures. A more extensive discussion can be found in [38, 13, 37].

7.1 Bounds checkers

[18, 32, 4, 16, 20, 25, 27] is a better solution to buffer overflows, however when implemented for C, it has a severe impact on performance and may cause existing code to become incompatible with bounds checked code. Recent bounds checkers [3, 41] have improved performance somewhat, but one major limitation of these bounds checkers compared to ValueGuard is that they do not detect buffer overflows in code that has not been protected even if the data is used in protected code. ValueGuard will detect changes to data even if it has been overwritten by unprotected code, as soon as the data is used in protected code.

7.2 Probabilistic countermeasures

Many countermeasures make use of randomness when protecting against attacks. Canary-based countermeasures [12, 14, 19, 28] use a secret random number that is stored before an important memory location: if the random number has changed

after some operations have been performed, then an attack has been detected. Memory-obfuscation countermeasures [11, 7] encrypt (usually with XOR) important memory locations or other information using random numbers. Memory layout randomizers [34, 6, 36, 8] randomize the layout of memory: by loading the stack and heap at random addresses and by placing random gaps between objects. Instruction set randomizers [5, 17] encrypt the instructions while in memory and will decrypt them before execution.

While our approach is also probabilistic, it is aimed at protecting locations from non-control-data attacks, while most of the above approaches are aimed at protecting either control data or preventing the attacker from injecting code, neither of which are useful for non-control data attacks.

An exception is DSR [7], which protects against non-control-data attacks but requires that all code is aware of the data obfuscation, hindering the use of third party libraries.

7.3 Separation and replication of information

Countermeasures that rely on separation or replication of information will try to replicate valuable control-flow information [35, 10] or will separate this information from regular data [39, 40]. This makes it harder for an attacker to overwrite this information using an overflow. Some countermeasures will simply copy the return address from the stack to a separate stack and will compare it to or replace the return addresses on the regular stack before returning from a function. These countermeasures are easily bypassed using indirect pointer overwriting where an attacker overwrites a different memory location instead of the return address by using a pointer on the stack. More advanced techniques try to separate all control-flow data (like return addresses and pointers) from regular data, making it harder for an attacker to use an overflow to overwrite this type of data [40].

While these techniques can efficiently protect against buffer overflows that try to overwrite control-flow information, they do not protect against non-control-data attacks.

7.4 Runtime enforcement of static analysis results

In this section we describe two countermeasures that provide runtime enforcement of results of static analysis.

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. This approach, while strong, does not protect against non-control data attacks.

WIT [2] discusses a very efficient technique to check whether instructions write to valid memory location. Their technique is based on static analysis that

does a points-to analysis of the application. This analysis is then used to assign colors to memory locations and instructions. Each instruction has the same color as the objects it writes to. Then runtime checks are added to ensure that these colors are the same. This prevents instructions from writing to memory that they cannot normally write to. This technique depends on a static points-to analysis, which can result in false negatives where an instruction is determined to be safe when it is not or it can assign an instruction or object a color that allows an unsafe instruction access to the object. Also, static alias analysis could confuse objects, allowing instructions access to multiple objects.

8 Conclusion

The increased difficulty of reliably exploiting control data attacks in modern operating systems is likely to shift the attention of attackers to other attack vectors. We believe that data-only attacks is such a vector since its successful exploitation can provide the attacker with as much leverage as traditional control-data attacks.

In this paper we presented ValueGuard, a countermeasure for data-only attacks caused by buffer overflows. ValueGuard's detection technique consists of inserting canary values in front of all memory objects and verifying them when the objects are used. Our countermeasure operates on the source code level and does not require any modifications to the target platform. In addition, ValueGuard can be used either as a testing tool by developers before deployment of an application or as a run-time protection monitor for critical applications.

Using ValueGuard we found a previously unreported buffer overflow in the Olden benchmark suite and we showed that ValueGuard can detect and stop data-only attacks that many other generic countermeasures cannot.

Acknowledgments

This research is partially funded by the Interuniversity Attraction Poles Programme Belgian State, Belgian Science Policy, and by the Research Fund K.U.Leuven.

References

1. Abadi, M., Budiu, M., Erlingsson, U., Ligatti, J.: Control-flow integrity. In: 12th ACM Conference on Computer and Communications Security (2005)
2. Akritidis, P., Cadar, C., Raiciu, C., Costa, M., Castro, M.: Preventing memory error exploits with WIT. In: IEEE Symposium on Security and Privacy (2008)
3. Akritidis, P., Costa, M., Castro, M., Hand, S.: Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors. In: 18th USENIX Security Symposium (2009)
4. Austin, T.M., Breach, S.E., Sohi, G.S.: Efficient detection of all pointer and array access errors. In: ACM Conference on Programming Language Design and Implementation (1994)

5. Barrantes, E.G., Ackley, D.H., Forrest, S., Palmer, T.S., Stefanović, D., Zovi, D.D.: Randomized Instruction Set Emulation to Disrupt Binary Code Injection Attacks. In: 10th ACM Conference on Computer and Communications Security (2003)
6. Bhatkar, S., DuVarney, D.C., Sekar, R.: Address Obfuscation: An Efficient Approach to Combat a Broad Range of Memory Error Exploits. In: 12th USENIX Security Symposium (2003)
7. Bhatkar, S., Sekar, R.: Data space randomization. In: 5th Conference on Detection of Intrusions and Malware & Vulnerability Assessment (2008)
8. Bhatkar, S., Sekar, R., DuVarney, D.C.: Efficient techniques for comprehensive protection from memory error exploits. In: 14th USENIX Security Symposium (2005)
9. Chen, S., Xu, J., Sezer, E.C., Gauriar, P., Iyer, R.K.: Non-control-data attacks are realistic threats. In: 14th USENIX Security Symposium (2005)
10. Chiueh, T., Hsu, F.: RAD: A compile-time solution to buffer overflow attacks. In: 21st International Conference on Distributed Computing Systems (2001)
11. Cowan, C., Beattie, S., Johansen, J., Wagle, P.: PointGuard: Protecting Pointers From Buffer Overflow Vulnerabilities. In: 12th USENIX Security Symposium (2003)
12. Cowan, C., Pu, C., Maier, D., Hinton, H., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P., Zhang, Q.: StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In: 7th USENIX Security Symposium (1998)
13. Erlingsson, U., Younan, Y., Piessens, F.: Low-level software security by example. In: Handbook of Information and Communication Security. Springer (2010)
14. Etoh, H., Yoda, K.: Protecting from stack-smashing attacks. Tech. rep., IBM Research Divison, Tokyo Research Laboratory (2000)
15. Jim, T., Morrisett, G., Grossman, D., Hicks, M., Cheney, J., Wang, Y.: Cyclone: A safe dialect of C. In: USENIX Annual Technical Conference (2002)
16. Jones, R.W.M., Kelly, P.H.J.: Backwards-compatible bounds checking for arrays and pointers in C programs. In: 3rd International Workshop on Automatic Debugging (1997)
17. Kc, G.S., Keromytis, A.D., Prevelakis, V.: Countering Code-Injection Attacks With Instruction-Set Randomization. In: 10th ACM Conference on Computer and Communications Security (2003)
18. Kendall, S.C.: Bcc: Runtime Checking for C Programs. In: USENIX Summer Conference (1983)
19. Krennmair, A.: ContraPolice: a libc Extension for Protecting Applications from Heap-Smashing Attacks (2003)
20. Lhee, K.S., Chapin, S.J.: Type-Assisted Dynamic Buffer Overflow Detection. In: 11th USENIX Security Symposium (2002)
21. Microsoft Coorporation: Detailed description of the Data Execution Prevention
22. Moore, D., Paxson, V., Savage, S., Shannon, C., Staniford, S., Weaver, N.: Inside the slammer worm. IEEE Security and Privacy 1(4), 33–39 (2003)
23. Moore, D., Shannon, C., claffy, k.: Code-red: a case study on the spread and victims of an internet worm. In: 2nd ACM Workshop on Internet measurment (2002)
24. Necula, G.C., McPeak, S., Rahul, S.P., Weimer, W.: Cil: Intermediate language and tools for analysis and transformation of c programs. In: Conference on Compiler Construction (2002)
25. Oiwa, Y., Sekiguchi, T., Sumii, E., Yonezawa, A.: Fail-Safe ANSI-C Compiler: An Approach to Making C Programs Secure: Progress Report. In: International Symposium on Software Security 2002 (2002)

26. OverTheWire: The OverTheWire hacker community. http://www.overthewire.org/
27. Patil, H., Fischer, C.N.: Low-Cost, Concurrent Checking of Pointer and Array Accesses in C Programs. Software: Practice and Experience 27(1) (1997)
28. Robertson, W., Kruegel, C., Mutz, D., Valeur, F.: Run-time Detection of Heap-based Overflows. In: 17th Large Installation Systems Administrators Conference (2003)
29. SANS: Top 25 Most Dangerous Programming Errors
30. Solar Designer: Non-executable stack patch (1997)
31. Spafford, E.H., Spafford, E.H.: The internet worm program: An analysis. Computer Communication Review 19 (1988)
32. Steffen, J.L.: Adding Run-Time Checking to the Portable C Compiler. Software: Practice and Experience 22(4) (1992)
33. Strackx, R., Younan, Y., Philippaerts, P., Piessens, F., Lachmund, S., Walter, T.: Breaking the memory secrecy assumption. In: 2nd European Workshop on System Security (2009)
34. The PaX Team: Documentation for the PaX project
35. Vendicator: Documentation for Stack Shield (2000)
36. Xu, J., Kalbarczyk, Z., Iyer, R.K.: Transparent Runtime Randomization for Security. In: 22nd International Symposium on Reliable Distributed Systems (2003)
37. Younan, Y.: Efficient Countermeasures for Software Vulnerabilities due to Memory Management Errors. Ph.D. thesis, Katholieke Universiteit Leuven (2008)
38. Younan, Y., Joosen, W., Piessens, F.: Code injection in C and C++ : A survey of vulnerabilities and countermeasures. Tech. Rep. CW386, Departement Computerwetenschappen, Katholieke Universiteit Leuven (2004)
39. Younan, Y., Joosen, W., Piessens, F.: Efficient protection against heap-based buffer overflows without resorting to magic. In: 8th International Conference on Information and Communication Security (2006)
40. Younan, Y., Joosen, W., Piessens, F.: Extended protection against stack smashing attacks without performance loss. In: 22nd Annual Computer Security Applications Conference (2006)
41. Younan, Y., Philippaerts, P., Cavallaro, L., Sekar, R., Piessens, F., Joosen, W.: PAriCheck: an efficient pointer arithmetic checker for c programs. In: ACM Symposium on Information, Computer and Communications Security (2010)