STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LVI**, Number 4, 2011

ENHANCING THE STACK SMASHING PROTECTION IN THE GCC

DRAGOŞ-ADRIAN SEREDINSCHI AND ADRIAN STERCA

Abstract. The paper addresses the problem of stack smashing or stack overflows in modern operating systems. We focus on a security solution for this problem, namely compiler generated canary protection and, to be more specific, we consider the Stack Smashing Protector (SSP) present in the most popular C compiler, the GCC. We first analyze the limitations of the GCCs SSP and then present three improvements that will harden the security offered by the SSP making an attackers attempt more difficult. All improvements refer to the most recent version of GCC, 4.6.2.

1. Introduction to Stack Smashing

The stack is an essential part of a running process (i.e. essential for implementing function calls), parts of it, called logical stack frames, correspond to every function which is being executed at a given moment; each time a function is being called, a new stack frame is created, and when the control flow returns to the caller, the related frame is popped out [11]. Smashing the stack is one of the most known and exploited variety of buffer overflow vulnerabilities, in the same time is considered to be the most readily treatable [4]. Even though attacks of this type are reported for over 25 years [12], the security patches, hardware and software protections seem to always be one step behind the latest type of exploit. Smashing the stack became a more popular issue once an interesting article with the title *Smashing the Stack For Fun and Profit* was published by Elias Levy (also known as Aleph One), in the Phrack online magazine [11] in 1996. This publishing did not only set off a multitude of system attacks, but, more importantly, raised the awareness of these problems and triggered the creation of defenses against them. Lets consider a simple C program which has a function named `func()`, that receives two

Received by the editors: November 5, 2011.

2010 *Mathematics Subject Classification.* 68Q01, 68Q01.

1998 *CR Categories and Descriptors.* D.3.4 [**Software**]: Programming languages – *Processors*; K.6.5 [**Computing Milieux**]: Management of computing and information systems – *Security and protection*.

Key words and phrases. stack smashing, canary protection, GCC, buffer overflow.

arguments and has as local variables, a character buffer `buf[80]` and an int. A depiction of the stack after the control flow reaches inside `func()` follows in Figure 1.

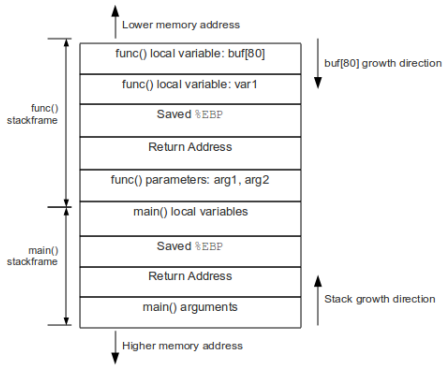

FIGURE 1. Representation of a standard process stack after entering the `func()` function

A simple unchecked `strcpy()` into `buf[80]` with more than 80 bytes will result in writing the excess data on the stack in neighboring places at consecutive higher memory addresses. Given that the stack grows towards lower addresses, overflowing `buf[80]` means altering the data beneath it (higher address), in this case: `var1`, the `Saved %EBP`, then the `Return Address`, and so on, based on how many surplus bytes were supplied. The `Return address` is saved on the stack whenever a function is called, in order to know where the control should return to (i.e. following instruction after the function call), so by changing this address an attacker can trigger the execution of arbitrary code when the vulnerable function finishes [11, 13]. In most cases, the data which is copied into `buf[80]` consists of an shellcode (i.e. machine instructions of code that typically spawns a terminal). By changing the `Return Address` to point inside this shellcode, as soon as the function is finished, the shellcode gets executed provided the privileges with which the process is run under are elevated enough to run those instructions and the address where the shellcode resides was correctly given (usually it is guessed). Other attacks involve changing the `%EBP register` [5], or altering some local data which is needed further in the function [5, 6].

2. Related work. GCC Stack Protection. Limitations

Leaving aside any hardware-enforced protections, which basically consist of a No Execute flag (NX) placed on certain areas of memory marking them as

non-executable, on the Linux family of operating systems, most of the progress that was obtained can be observed at two particular levels:

- GNU Compiler Collection
- Kernelspace

First of all, the compilers defense against exploits we are considering can be divided into two subcategories [1]:

(1) Array bounds checking - the design of the C language makes this method unachievable without compromising either the performance or the compatibility, as an example see the *Bounds Checking Project* [14].
(2) Data integrity checking - the principle is simple: guard the sensitive data using some predefined value and detect changes (corruptions) to the guarded data by checking the predefined value. Achieved through the StackGuard project [2], which was reimplemented in GCC under the name of Propolice, but generically referred to as Stack Smashing Protector (SSP).

Security enhancements were added in several steps to the GNU Compiler Collection, increased abstractization was desired in order to achieve a wider platform independence, and the most notable achievements that are now part of this compiler system are:

- *GNU_STACK* ELF Markings - A stack-controlling ELF program header, used to emulate the behavior of the NX bit. By default this is disabled, no marking being inserted, so, if no GNU STACK note is found, it is assumed that the stack needs to be executable (this might be needed for trampolines) [3].
- *D_FORTIFY_SOURCE* flag - Compiler flag that inserts compile and run-time checks for certain unsafe libc functions. Enabled whenever the optimization level is set to a value bigger or equal with 2 (-O2).
- *Stack Smashing Protector (SSP)* - An extension to the compiler based on the StackGuard security model, that is, guarding the stack frames of a process through a known value placed in certain areas on the stack. The discussion on the protections and flaws of this mechanism is done in the next subsection.

2.1. **SSP Protection Model.** Firstly, it is necessary to point out that the contents of this section refer to version 4.6.2 of the GCC which is the latest version up to the date of this writing. The idea behind the design of this protection lies in the fact that whenever a buffer overflows, all adjacent memory areas up to a certain point are overwritten, and by placing a known value just after a buffer, if an overflow takes place that value will be altered. By doing

a simple check to see if the value remained unchanged it can be deducted whether the guarded buffer has overflown or not. The value pushed on the stack is named a canary, and there are two types that are mainly used:

- **Random Canary** - the value of the canary will be random and the rationale behind this is to avoid the situation in which an attacker knows (or guesses) the value of the canary and fills the overflowing bytes with this value, thus when the canary state is checked it will not be detected as corrupted, but an overflow still took place.
- **Terminator Canary** - a canary with the value `0x000AFF0D` [4]. The significance of this special value stands in the fact that an attempt to overflow the buffer and preserve the canary value unchanged (thus, writing at some point `0x00`) will stop the `strcpy()` C function and, the same effect has `0x0A` on `*gets()` [5], `0xFF` represents EOF, and `0x0D` is the code for carriage return, used as well in order to stop string manipulation or copying functions.

As a side note, other two types of canaries exist: *Null Canary* - holding the null value, which was given up on, and the *XOR Random Canary* - represented by a random number which is xor-ed with the return address saved on the stack [5]. The SSP protection does not resort to any of these two canary types, so we shall not insist on them.

The location where the canary is placed, whatever its type is, is essential for the design of this defense mechanism. Initially - in StackGuard - the canary was placed between the `saved %EBP` and the `Return Address`, but in SSP the canary code resides above both `%EBP` and `Return Address` (remember that above on the stack in this context means being at a lower address because the stack grows from bigger to lower addresses) and below the local variables. In this way, if one of the variables overflows the stack it will inevitably change the canary before the overflow gets to `%EBP` or the `return address` of the function (see Figure 2).

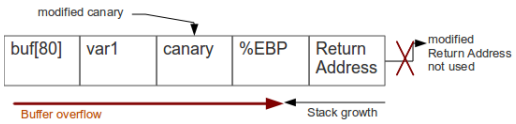

FIGURE 2. Canary placement on the stack

The canary protection code is automatically inserted in an application by the compiler at compile time. For each call to a user defined function two types of actions are performed:

- *Canary Insertion*: The value of the protective canary is inserted in one of the compilation phases, more precisely, at the function prologue level (i.e. entry-code).
- *Verification*: Canary state is verified for consistency before the data that the canary is meant to protect is used, that is, before the stack frame is discarded - action undertaken in the Return Code (i.e. epilogue level).

As said earlier, each protection was meant to stop one particular aspect of an exploit, the canary mechanism's job being to guard the important values of the `return address` and the `saved EBP`. However, together with the canary protection, other two important security enhancements were added to GCC along the way:

(1) *Reordering of function local variables*: If it is written in a local buffer more than its declared size, some other local variables will be altered before reaching the canary; moreover, in some cases the canary is not reached at all, and the intention is to only overwrite some local variable that is used elsewhere - the attacker could be after a format string exploit for example (more can be read in [6] or [5]). Thus, the variables are ordered respecting a single criteria: buffer or not. The buffers are all arranged adjacent to the canary, successively, and the other non-buffer locals will reside above the buffers - in this way if any overflow happens, only other arrays may be corrupted, no local data is altered [7]. A flaw in this type of protection can be observed here in the fact that not only non-buffers should be protected from overflow corruption, but also buffers that might hold important information. By reordering the local data in this way, the buffers are left to overflow into each other, posing a serious security risk to the application.

(2) *Copy of function arguments above the local variable buffers*: Through this, whenever inside a function the arguments are needed, they will no longer be taken from below the `%BP` register, but from an area preceding the local data of this function. Done in the first moments after entering the function, after the creation of the stack frame, each argument is copied from `%EBP+8`, `%EBP+12` etc., on top of the newly created frame. The logic behind this mechanism is the following: during the execution of instructions from the function, its arguments can be corrupted (along with the canary, `%BP` etc.) and then used all throughout that function, even passing them further to other subroutines. The canary state is only checked at the moment when the function returns, which is too late, because probably the execution flow was already corrupted. The parameters are so copied and they will precede the

buffers which could overflow [7], and the possibility of using corrupted data is solved.

Illustration of all the three SSP mechanism for protecting against stack-smashing are presented in Figure 3.

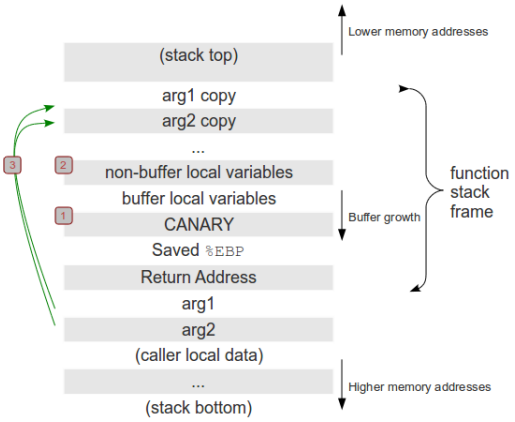

FIGURE 3. SSP Protection added on the stack: canary (1), reordering of local variables (2) and copying of the function parameters (3)

3. IMPROVEMENTS TO THE GCC STACK SMASHING PROTECTION

Since its beginning in 2001, when it was introduced as a FreeBSD GCC protection patch [7], several improvements were consecutively added to SSP to fix newly discovered ways of compromising the stack. The fact that limitations and flaws can still be observed along with the existence for some time of attacks that can bypass SSP [5], stand as proof that the system needs further development and maintenance. Based on the observed limitations and the evolution of SSP, a final version of SSP which would fix all possible security holes seems very hard to find, and, just as it is seen in all aspects of computer security, simply going on and continually hardening and limiting the attacker possibilities might be the best solution.

In this section we present three proposals for increasing GCC stack smashing protection and hardening a hackers effort to compromise the target machine. All our proposals refer explicitly to the most up-to-date, stable version of the GCC compiler which is, at the time of this writing, 4.6.2 [10]. They are intended to complement existing security protection mechanisms present in GCCs SSP (i.e. canary protection, variable reordering, and argument copying).

3.1. **Improvement #1 - Canary state verification not only when the function returns, but also when the function issues a call to another function.** The idea behind this proposal is that the canary signalizes a corruption, but the moment when the value is checked is essential. Currently, this check is performed only in the function epilogue (i.e. when the functions stackframe is released and the control is returned back to the caller through the Return Address) because the initial intention was to provide a protection to the saved `%EBP` and `Return Address` only - which are only used at this point in the execution flow. But, as it can be seen, not only the `Return Address` and the saved `%EBP` are corruptible. Even with the local variables reordered, a possible attacker can still overflow local buffers or variables (i.e. buffers and variables declared by the current function) or structure members can be corrupted by arrays from inside those structures (because inside a structure variables are not reordered by SSP) and these local buffers, variables or structures might be passed in subsequent function calls. The idea behind adding this enhancement lies in the fact that, applications compiled with the SSP protections do not benefit from corruption detection throughout a function code, only at the function end, and at that moment it could be too late, given that local data could have been altered and passed to other function, then used in a chain of exploits.

A method of avoiding the aforementioned problems is to check the canary of the caller function when the execution is about to enter in a new subroutine. In order to do this, there are two ways, or possible points, where the new verification can be placed:

(1) The callers canary could be checked to see if it is unchanged in the prologue of the called function. Doing this check here makes sense because canary operations (i.e. canary placement on the stack and canary verification) are performed in the GCC in a functions prologue and epilogue so in order to keep the injection code clean and compact the check of the callers canary should be performed either in the called functions prologue or epilogue and the called functions prologue is the closest point in execution time to a possible stack corruption in the caller function. However there is an important drawback of doing the callers canary check at this point, in the called function, because the prologue of the called function must be aware of the structure of the callers stackframe (more specifically, the location of its canary) which is normally isolated from the current (i.e. called functions) stackframe.
(2) Another place where the canary can be checked is when a call to a new function is issued, before issuing the call and creating the stackframe for the called function. The advantage of this approach is that the

corruption is detected a bit earlier than in the first case and there is no need of violating the stackframe isolation between function calls, but it also adds complexity to the SSP code making it less compact, as SSP code is normally inserted only in a functions epilogue or prologue, not when a call to a new function is issued.

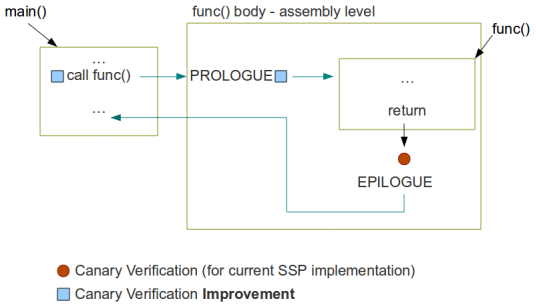

FIGURE 4. The Verification of the canary value done at multiple steps (the rectangles represent the two best positions where the new verification should be inserted)

A note regarding Figure 4: The rectangles represent points where the verification of the canary of the `main()` function are to be done, not `func()`; the canary for `main()` is inserted automatically by the compiler, `main()` being called from the global initializer (constructor) function localized in `__init` (see Section 17.21.5 of [1]).

This improvement mainly aims at doing a preemptive canary corruption detection by simply adding another verification of its state. The overhead should be considered especially for applications that are very modular and for which the flow of execution is expected to pass through many user-defined functions. Particular attention also needs to be given for programs that rely on recursive and highly recursive functions, for which this protection should probably not be used. The targeted set of applications upon which multiple canary verification will have the best impact are the special-purpose daemon services like `crond, ftpd, sshd, init, postfix` or `sendmail` that are critical for any Unix machine, very rarely recursive calls being found in these programs and their security is vital for the system defense. The overhead is comparable with the one for the current SSP canary verification step in the epilogue, so it should not be regarded as an drawback only in special cases as the ones we have mentioned above.

3.2. **Improvement #2 - Canaries for every individual buffer.** One of the obvious flaws of SSP is that the buffers are grouped together, consecutively,

thus enabling situations where one buffer could overflow into another without the canary even being reached and corrupted. This imperfection might seem somewhat harmless in the context where buffers are only used to hold data in transit (even in this case, the data is corrupted when a buffer is overflown into), but it poses a great security risk once the importance of that data increases (i.e. this data is passed to other functions). Take for example the registration of an user account for a fictive application, the password string will be kept in a buffer and if an overflow of other array into that buffer takes place, it will result in storing the corrupted password for that user. A small overflow of only a few bytes (such that the canary is not reached) will pass unnoticed by the current SSP protections, so scenarios like the one described earlier are possible, and they should not be overlooked.

Our second improvement to SSP implies using a different canary for protecting every individual local buffer of a function. In this approach, a canary space (i.e. 4 bytes) is reserved on the stack in front (i.e. higher address) of every local buffer (see Figure 5). Individual canaries for buffers will extend the functionality of SSP from guarding sensitive, controlling data (`%EBP` and `Return Address`) to guarding every vulnerable buffer of the program. This way the task of an exploiter will be even further hardened.

But this added security comes with an important overhead cost. The added overhead cost (i.e. canary generation, storing and verification) of having individual canaries for every buffer is proportional to the number of local buffer variables used in a function and is not negligible for applications dealing with large amounts of data (e.g. network applications, multimedia applications etc.).

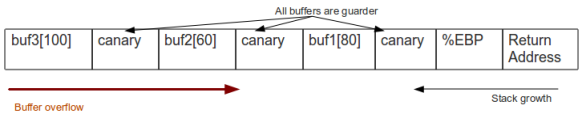

FIGURE 5. A different canary protects every buffer, not just the `Return Address` and `%EBP`

To be noted here that in our 2nd improvement each buffer is protected by its own canary, but still this approach shares the vulnerabilities that a regular canary check has (e.g. the byte-by-byte technique of canary discovery), just that it makes it more difficult to discover the canary value, because now we have several canaries.

Because using unique canaries for individual buffers incurs important additional overhead, this technique can be adapted to offer a good compromise between overhead and protection. For example, the compiler can decide at

compile time, based on its own estimations or based on user input through compiler options, how many canaries to use and in this approach, one canary could be used to protect several local buffers; thus, the protection achieved is less than when using unique canaries for individual buffers, but it has a smaller performance cost.

3.3. **Improvement #3 - Probabilistic canary protection.** The next defense suggestion aims at significantly hardening the process of uncovering the canary value. It does this by two distinct methods, methods which are supposed to be used in conjunction:

- canary position randomization
- probabilistic failure in case of canary corruption detection

The most important protection is given by the second method, the first being just a complementary protection for the second method. Prior to introducing these two methods, we must first briefly discuss the byte-by-byte technique of canary disclosure. This is done in the following paragraph.

The byte-by-byte canary value disclosure technique [8]

This method uses the fact that if child programs are spawned using `fork()`, then the terminator canary will be shared among the parent process and every child that it creates. `fork()`-ing is a crucial feature supported by the operating system, and it is used by an important set of applications that are essential for any Unix distribution. Network deamons usually create handling processes for each connection in this way, and the presence of this flaw in this kind of applications poses a great security risk for a computer exposed to the Internet. The technique is based on the fact that it is easier to guess the value of each single byte of the canary individually (0..255 possible values) rather the value for the whole 32-bit canary at once (0..4294967295 possible values). Consequently, the byte-by-byte technique first overflows the buffer with only 1 byte, thus overwriting the first byte of the canary. When the SSP detects canary corruption it causes the application to crash fast. If on the other hand, the first byte of the canary is overflowed with the correct value, the application will continue its execution (i.e. will not crash). since there are only 256 possible values for a byte, the attacker can try successive values for that byte until the value is found and in the worst case scenario, it needs 256 successive trials until the value of the canary byte is discovered (i.e. the overflowed byte for which the application does not crash). After the first canary byte is discovered the next byte is overflowed in order to discover its value and so on until all 4 bytes of the canary are discovered.

So the whole technique consists of successively trying values for every byte of the canary until the value is found [8], first only one byte is overflowed, then two bytes, and so on. The maximum number of trials an exploiter has to do is $256 \cdot 4 = 1024$, which, in fact, is not even that much for a networking daemon such as `httpd` for example, each individual try taking place in one child of the daemon process. Once the whole canary value was compromised, the protection itself for that process can be considered as nonexistent, the same canary being used for any function and for any child process. The whole process is depicted in Figure 6.

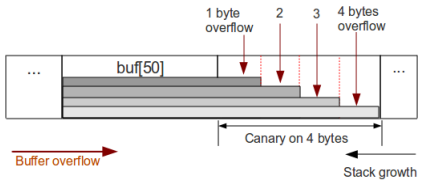

FIGURE 6. Multiple overflows of 1 byte will result in finding the first byte of the canary. Same idea is applied successively 4 times, for each individual byte

Currently, the canary position is fixed, it will reside three stack entries (remember that a stack entry is the same as a word - 4 bytes, on 32 bit architectures) above the `%EBP`, thus leaving 2 empty words, and right next to the first declared buffer. If that buffer overflows even with only 1 byte, the canary value will change, this being the reason why the canary and the buffer are sticked together. But having this specific information at his disposal an attacker can and will use it. By knowing the exact position on the stack of a canary the amount of work someone has to do at finding its value is halved. If the position were to be unknown, he would have to first find that position and than proceed to searching the value.

The first part of improvement no. 3 is canary position randomization. By setting up an area of 10 words space between the `Saved %EBP` and the first buffer and randomizing the position of the canary in this region, the canary value search technique is hardened, first having to find the position of the canary and then proceed to trying the byte-by-byte technique. The overhead added by this step is insignificant, the position could be established using a simple random number generator (e.g. kernels /dev/urandom in Linux). Ten stack positions used for every function in the current stacktrace is, likewise, not an important penalty from the memory point of view.

However, on a closer look, the same method of overflowing byte-by-byte can still be applied to finding the position. So, in fact, in this way only a few

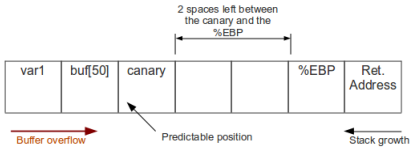

FIGURE 7. The position of the canary is fixed, lying at an predictable address relative to the buffers and the stack frame

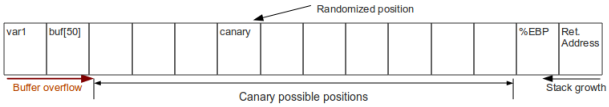

FIGURE 8. The position of the canary is randomized in an area of 10 stack entries

more tries are needed by the attacker, a maximum of 10, added to that of 1024, the results would be unsatisfactory. Therefore, along with randomizing the position of the canary, another method is required to make the process of canary value disclosure more difficult. This method is the *probabilistic failure in case of canary corruption*.

The basis of the probabilistic failure in case of canary corruption method is the observation that in the byte-by-byte technique, an attacker relies on a binary feedback from the SSP code when a byte of the canary is overwritten: failure/crash of the application meaning the value of the overwritten canary byte is incorrect and non-failure (i.e. successful execution) meaning the value for the overwritten canary byte is correct. If we somehow manage to make the SSP code still give an attacker a feedback in a binary form (i.e. failure/non-failure of the application), but the interpretation of a specific feedback is multi-valued and uncertain (i.e. it is not certain that a non-failure means the value for the overwritten canary byte is correct, it might just as well be incorrect), then the byte-by-byte technique will have a hard time guessing the canary. And a good source of uncertainty is randomness. Following this line of thinking, we want to modify SSPs behavior so that it does not crash the application every time the canary or a part of the canary gets corrupted. This is why we call this method probabilistic failure in case of canary corruption. If the application is crashed by the SSP on a random basis when a part of the canary is corrupted, the attacker can no longer infer whether he has guessed that part of the canary or not. The maximum that can be obtained from any canary disclosure protection method is to force an attacker to make 2^{32} = 4294967295 successive trials before he discovers the canary value. And this is exactly the goal of our probabilistic failure method - to force an attacker

to run a number of canary trials close to 2^{32}. Of course, if the canary is not corrupted at all the SSP should not crash the application because something like this will break the compatibility with almost all the applications which exist nowadays. Similarly, if the corruption has moved beyond the canary to the `%EBP` and `Return Address` (i.e. the last byte of the canary is corrupted), the application should necessary crash, because a non-failure will essentially remove all the protection offered by stack canaries.

Thus, our method of probabilistic failure in case of canary corruption will be implemented in the epilogue of a function where canary corruption verification takes place. If no byte of the canary is corrupted, the execution continues normally, failure-free. If the last byte of the canary (i.e. the byte closest to the `%EBP` and `Return address`) is corrupted, the execution crashes. In any other situation (i.e. when a part of the canary is corrupted) our method will compute a failure probability, P_{fail} and based on that, it will crash the application or execute successfully. The algorithm is described bellow:

```
if (last byte of the canary is corrupted)
        fail();
else if (no byte of the canary is corrupted)
        continue;
else
        r = random(0,1);
        P_fail = 1/e^n · r;
        if (P_fail >= 0.5)
            fail();
        else
            continue;
        endif
endif
```

In the above algorithm n is the number of modified bytes in the canary and $random(0, 1)$ is a function which generates a random number between 0 and 1. $fail()$ is a system call that immediately ends the current application returning an error code and *continue* continues the execution of the current application.

By multiplying the random number r with $\frac{1}{e^n}$ the failure probability formula we want to have a higher probability that the program will not crash when more bytes from the canary are corrupted. This is because we want more ambiguity when more bytes of the canary are tried by the attacker. Because of this probabilistic failure method, when an attacker overflows a new byte of

the canary and the program does not crash he can not conclude with certainty that he has discovered that canary byte.

Together with canary position randomization, the probabilistic failure in case of canary corruption method makes it significantly harder for an attacker to discover the canary value. In order to assess the overload induced by our improvement number 3 to the running code, we have implemented this improvement in the GCC's SSP and run 4 tests using 2 test environments. In the first test environment we have a simple program in which the `main()` function calls a simple function, `func()`, several times, first only one time, than ten times, then one hundred times and finally one thousand times. In each test, we have recorded the time (in microseconds) it takes to call function `func()` the specified number of times with the improvement patch enabled and without the improvement patch enabled. The measured times are depicted in the following table.

	Without improvement no. 3 patch	With improvement no. 3 patch
main() calls func() 1 time	2 μs	2 μs
main() calls func() 10 times	3 μs	3 μs
main() calls func() 100 times	4 μs	4 μs
main() calls func() 1000 times	19 μs	23 μs

For the second test environment, we have the same setup as in the first test environment, but this time, each time is being called, function `func()` calls another function `func_1()`. In each test, we have recorded the time (in microseconds) it takes to call function `func()` the specified number of times with the improvement patch enabled and without the improvement patch enabled. The measured times are depicted in the following table.

	Without improvement no. 3 patch	With improvement no. 3 patch
main() calls func() 1 time	2 μs	2 μs
main() calls func() 10 times	3 μs	3 μs
main() calls func() 100 times	5 μs	6 μs
main() calls func() 1000 times	34 μs	38 μs

It can be seen from both tables, that the improvement number 3 does not pose significant overload cost for the running code.

4. Conclusions

This paper presented 3 improvements to the Stack Smashing Protector (SSP) mechanism from the GCC compiler (most recent version being GCC 4.6.2), improvements which would harden the security of applications and will make an attackers job more complicated. The first improvement relied on performing the canary verification step as close as possible to the moment when the actual overflow happened. Improvement number 2 deals with unique canaries for each local buffer of a function and lastly, the most important improvement, probabilistic failure in case of canary corruption aims at making the canary discovering task a difficult one. The GCC patch containing the code for the probabilistic failure method can be downloaded from [9].

5. Acknowledgments

This work was partially supported by CNCSIS-UEFISCSU, through project PN II-RU 444/2010.

References

[1] *** *A GNU Manual*, http://gcc.gnu.org/onlinedocs/gccint/index.html, Free Software Foundation, Inc, 2010.
[2] Crispin Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qiang Zhang and Heather Hinton, *StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks*, pages 63-77, 7th USENIX Security Conference, 1998.
[3] Mike Frysinger, solar, *The GNU Stack Quickstart*, http://www.gentoo.org/proj/en/hardened/gnu-stack.xml?style=printable
[4] Perry Wagle, Crispin Cowan, *StackGuard: Simple Stack Smash Protection for GCC*, GCC Developers Summit, 2003.
[5] Gerardo Richarte, *Four different tricks to bypass StackShield and StackGuard protection*, 2004, http://www.coresecurity.com/files/attachments/StackGuard.pdf.
[6] *** *Format String Exploits*, www.acm.uiuc.edu/sigmil/talks/general exploitation/format_strings/.
[7] Hiroaki Etoh, *GCC extension for protecting applications from stack-smashing attacks*, 2005, http://www.trl.ibm.com/projects/security/ssp/.
[8] Ben Hawkes, *Exploiting OpenBSD*, 2010, http://sota.gen.nz/hawkes openbsd.pdf.
[9] http://www.cs.ubbcluj.ro/ forest/research/grants/pd-444/gcc-ssp
[10] *** *The GNU C Compiler homepage*, http://gcc.gnu.org/
[11] Aleph One, *Smashing The Stack For Fun And Profit*, Phrack #49 article 14, 1996, http://www.phrack.org/issues.html?issue=49&id=14#article.
[12] Michael Dalton, Hari Kannan, Christos Kozyrakis, *Real-World Buffer Overflow Protection for Userspace & Kernelspace*, 17th USENIX Security Symposium, 2008.
[13] J. Reynolds, *RFC 1135 - The Helminthiasis of the Internet*, 1989.
[14] *** *Bounds Checking Project*, abandoned GCC project, http://gcc.gnu.org/projects/bp/main.html

Babes-Bolyai University, Department of Computer Science, 1 M. Kogălniceanu St., 400084 Cluj-Napoca, Romania
E-mail address: sdsd0494@scs.ubbcluj.ro
E-mail address: forest@cs.ubbcluj.ro