Preventing brute force attacks against stack canary protection on networking servers

Hector Marco-Gisbert, Ismael Ripoll
Instituto Tecnológico de Informática. Universitat Politècnica de València
Camino de Vera s/n, 46022 Valencia, Spain
{hmarco,iripoll}@iti.upv.es

***Abstract*—The buffer overflow is still an important problem despite the various protection methods developed and widely used on most systems (Stack-Smashing Protector, ASLR and Non-eXecutable). Most of these techniques rely on keeping secret some key information needed by the attackers to build the exploit. Unfortunately, the architecture of most web servers allows attacker to implement brute force attacks that can be exploited to obtain those secrets by mean of brute force attacks, and eventually break into the server.**

We propose a modification of the stack-smashing protector (SSP) technique which eliminates brute force attacks against the canary. The technique is not intrusive, and can be applied by just pre-loading a shared library. The overhead is almost negligible.

The technique has been tested on several web servers and on a complete GNU/Linux distribution by patching the standard C library. We expect that the strategy presented in this paper will become a standard technique on both desktop and servers.

I. Introduction

A decade ago, buffer overflows, specially stack smashing, was the most dangerous thread to computer system security. Over the last years, several techniques have been developed to mitigate the ability to exploit this kind of programming faults [1], [2]. StackSmash Protector (SSP), Address Space Layout Randomization (ASLR) and non-executable stack (NX) are widely used in most systems due to its low overhead, simplicity and effectiveness.

Following the classic measure/counter-measure sequence, a few years after the introduction of the each protection technique a method to bypass, or reduce its effectiveness, was introduced. The SSP can be bypassed using brute force or by overwriting non-shielded data [3], [4]; the ASLR can be bypassed using brute force attacks [5]; and the NX, which effectively blocks the execution of injected code, can be bypassed using ROP (Return Oriented Programming) [6]. In spite of the existing counter-measures, those techniques are still effective protection methods, in some cases they are the only barrier against attacks, until the software is upgraded to remove the vulnerability.

Unfortunately, the forked and pre-forked networking servers architecture are specially prone to brute force attacks. All the children processes inherit/share the same memory layout and the same canary than the parent process. The attacker can try in bounded time all the possible values of canary (for SSP) and memory layouts (for ASLR) until the correct ones are found. There is a very dangerous form of SSP vulnerability, called byte-for-byte, which allows the attacker to try each byte of the canary independently, which allows to find the value of the canary with just a few hundreds of trials (the system is defeated in seconds).

We present a modification of the SSP technique which consist on setting a new random value of the canary for each child process when the `fork()` system call is invoked. The technique is called RAF SSP (Re-new After Fork SSP).

Re-randomise the canary has not been seriously considered [4] mainly due to two factors: first, the protection increase is only by a factor of 2, on average, when compared with the standard SSP on a system with no other protection techniques. And second, the complexity of the implementation would not worth the protection improvement.

The RAF SSP technique greatly increases (several orders of magnitude) the difficulty of an attack when the three protection techniques (SSP+ASLR+NX) are employed, as it is the case in most systems. Regarding the problems that may cause a canary change on a running process, we have identified that the error confinement that represents each forked thread is also a de-facto stack confinement which allows to change the value of the reference canary with no impact on the correct operation of process.

A. Benefits of our proposal

The main properties of the RAF SSP are:

- It can be used just pre-loading shared library. There is no need to modify the source code of the networking servers, nor recompile the server, nor modify system libraries nor the compiler.
- It **prevent brute force attacks** against canary stack protection mechanism.
- The SSP byte-for-byte attack is no longer applicable to the RAF SSP.
- Multiplies the effectiveness of the combined protection of the SSP, ASLR and NX techniques. On a standard 32bit system the cost of breaking a system **takes 512 times more trials**, on average (from 2^{23} trials to 2^{32} on a 32bit system).
- The overhead introduces is negligible.

The RAF SSP is specially useful for networked server, but it is not limited to them. We have tested it on a complete Linux distribution by modifying the standard C library, with full functionality and no appreciable performance penalty.

II. Background & assumptions

A. Network server architectures

Network server software architectures has been an extensively studied and analysed due the importance of the web servers in the current network infrastructure. Attending to the processing model, we will focus on two basic models, paying special attention to the robustness of each approach. A complete list of the models is out of the scope of this paper (see [7]).

Multi-thread: each connection is handled on a dedicated thread. Multiple clients can be attended concurrently with a low overhead. The main drawback is that a crash of any thread may kill the whole server or let the server in an inconsistent state. There is a single error confinement region: the server process.

Multi-process: each connection is handled by a separate (child) process. There are two variants: *forking server*, where a child process is created explicitly to serve each request. As soon as the client request is finished the child process exists. The server process is able to attend client requests at any time, except while the child is being created (while forking). The other variant is the *pre-forking server*. Several sub-processes are forked before any connections are handled (when the server starts). Each child blocks for a new connection, handles the connection and then waits for the next connection. This removes the overhead of the `fork()` call at the time of accepting a new connection. The crash of a child process has a limited impact on the operation of the server, only the client that caused/suffered the crash have a failure. The error confinement region for this mode is isolated of each process.

A good compromise between performance and robustness (error confinement) is the hybrid multi-process multi-threaded used in Apache.

B. The stack smashing protection SSP

The first proposal was presented in [8] and improved over the years. Without loss of generality, we will assume that the stack grows downwards, i.e. to lower addresses. We will use the x86 architecture in the examples.

The SSP technique [9] is a compiler extension which adds a guard (the canary) between the protected region of the stack and the local buffers. Initially, the canary was placed right after the return address since it was the target of most attacks. Over the years, new attack strategies where developed (see section III) which motivated some enhancements [10]. As of GCC v4.6.3 the stack-smashing protector consists of:

- Both, the return address and the saved stack frame pointer are guarded by the frame canary.
- Local variables are reordered so that buffers are located first (higher addresses) and below them the scalar variables and the saved registers. This way, buffer overflows (which typically grows upwards) will not overwrite scalar variables.

Figure 1 sketches the layout a stack with two function frames. The reference-canary is represented as a small bird on the right side, and the frame-canary is the bird on each frame. The compiler emits extra code in the prologue and epilogued of each protected function for initialising and checking the value of the canary respectively.

The value of the canary is chosen such that: prevents, when possible, the effective exploitation of a buffer overflow; and detects the occurrence of an overflow. Attending to these, two (XOR canaries are not included because they have more

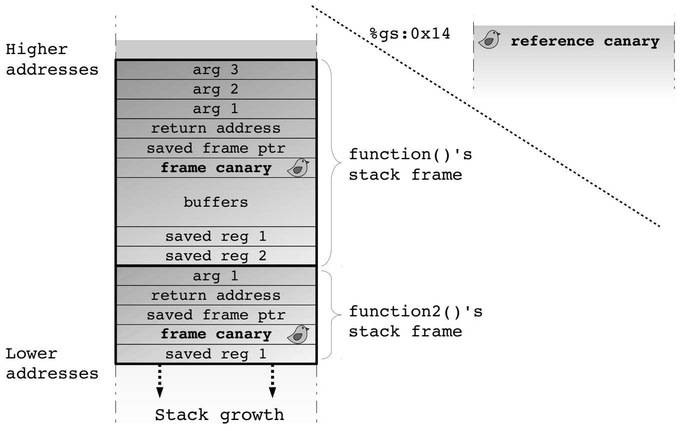


Fig. 1. x86 Stack layout.

overhead that the other types and the same properties as random canaries) kind of canary values has been proposed:

- Terminator value: the value is composed of different string terminators (CR, LF, NULL and -1).
- Random value: the value is a random value selected during the process initialisation. The attacker needs to know the actual value for building the exploit. As far as the value is kept secret, the attack will be prevented.

In most implementations, the canary value is a word with all bytes random except one that is zeroed.

Since the value of the canary is not a constant but a random value chosen when the program starts, that value has to be stored somewhere in the program memory (Or in a dedicated processor register if available). In the x86 and x86-64 architectures the reference-canary is stored in special data segment which is not accessible as a "normal" variable and can be not overwritten or read abusing any data structure.

The numerical examples given along the paper are referred to a GNU/Linux x86 architecture.

III. Threats

In this section, rather than a detailed explanation on how to bypass the SSP, we will present only the weaknesses of the stack canary that enables the possibility of an attack. In [11], the author explains the process to remotely exploit a buffer overflow on systems equipped with these techniques.

Basically, there are two ways to bypass the canary:

1) Overwriting the target data (return address, function pointer, etc.) without needing to overwrite the frame-canary,
2) Overwrite the frame-canary with the correct value.

Obviously, the data that is not guarded by the canary (exception handlers, function pointers in data structures, etc.) are prone to other forms of buffer overflow, but since our technique does not increase the coverage (detection capability) of the basic canary technique but reinforces the protection of the already protected items. We will focus on the brute force attacks against the canary value.

The second way to bypass the canary requires that the attacker knows the actual value of the canary. Processes created with `fork()` are a duplicate of the calling process. Both, father and child have the same canary value. On a forked server, where the service is attended by children of the server process, an attacker can build **brute force attacks** by guessing the value of the canary as many times as needed.

Depending on the granularity of how the attacker can overflow the buffer (word or byte overflow), there are two different brute-force attacks: *full brute force*, and *byte-for-byte*.

A. Full brute force attack

The frame-canary word is overwritten on each trial. If the guessed word is not correct then the child process detects the error and aborts. As consequence, the attacker does not receive a reply, which is interpreted as an incorrect guess. The guessed value is discarded, and attacker proceeds with another value until all the possible values are guessed.

On most 32bit systems the canary (word) has 3 random bytes plus one zeroed. In the worst case, the number of trials is 2^{24}; and $2^{23} = 8388608$ on average. This figures may deter a remote attacker but not a local attacker, which may break the system in a few hours.

B. Byte-for-byte brute force attack

If the attackers have a fine-grain control over the number of bytes that overflows then a byte-for-byte attack can be constructed. The attack consists on overwriting only the first byte of the canary until the child does not crash. All the values from 0 to 255 are tested sequentially until a success is got. The last byte tested is the first byte of the canary. The remaining bytes of the canary are obtained following the same strategy.

This kind of bug is very dangerous because a system is broken with only $3 \times 256 = 768$ trials. For this reason, most canary implementations set to zero one of the canary bytes (the most significant in x86) for preventing the bfb attacks when the overflow is performed by a string copy functions.

IV. PROPOSED STRATEGY

The following section first describes the strategy idea. Below an example of stack evolution is described to clarify the strategy. Finally a brief of some special cases where the canary reference should be changed carefully.

A. Observations

Our proposal is based on the following observations:

Observation 1: Most applications, specially networking servers, after a `fork()` operation the child process executes a flow of code which ends with an explicit call to the `exit()` system call. That is: the child process does not return from the function that started the child code.

We have validated this observation, both 1) analysing the code of several servers and 2) empirically by running a complete GNU/Linux distribution where processes that returns after a fork are killed.

Observation 2: Each child process of network server defines an error confinement region. That is, any error that occurs to a child will not affect the correct operation of the father and siblings processes.

Observation 3: There is a single reference-canary per process which is stored in a protected area and initialised during the process start up. It is copied in each stack frame between the saved stack frame and the buffers.

Observation 4: The integrity (compare the frame-canary against the reference-canary) is only done at the end of each function, right before the returning instruction.

Observation 5: Only the value of frame-canary of the current stack is checked against the reference-canary.

B. Renew canary at fork (RAF SSP) strategy

The renew canary at fork (RAF SSP) strategy consist in renew the value of the reference-canary of the child process right after it is created (forked). The new value is also a random value. Every child process have a different reference-canary.

From observation 1 we know that the child code would not return and so the frame-canaries stacked will never be tested, observation 5. Therefore, they do not need to be updated.

Although it is not difficult to check that a function never returns, most compilers provides the `noreturn` function attribute which declares a function as non-returning. The compiler generates more efficient code and checks (at compile time) whether the function honours the desired behaviour or not. It is advisable that the function where the canary is renewed has this attribute.

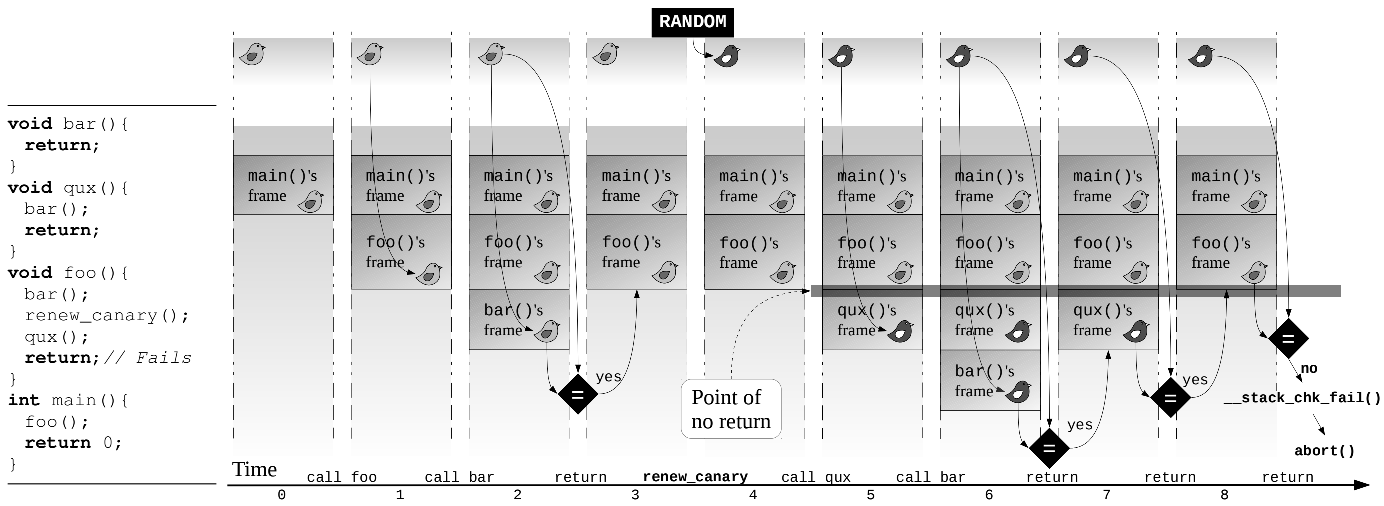
When the attacker guesses an incorrect value, the child is killed by the stack protector detection mechanism and a new child with a new canary is started. As a result, **brute force attacks can not be built**.

C. Illustrative examples

Example 1: Figure 2 represents the evolution of the stack of code on the left of the figure. Different canary values are represented by different colours. When a function is called, the stack frame is setup coping the reference-canary into the frame-canary (see stack state at time 1). Upon return, the frame-canary is compared with the reference-canary, which is represented by a black diamond with an equal sign inside. The `renew_canary()` operation changes only the reference-canary, the stack is not modified (from time 3 to 4), and it defines a point of no-return. The current function (`foo()` at time 4) can not return since there will be a mismatch between the reference and the frame canaries.

Note that the frame-canary of the previous stacked frames keep the old reference-canary value while the new frames have the new canary (times 5, 6 and 7). As far as the process never returns from those functions it would not a problem. Also, note that the same function (`bar()` on times 2 and 6) have different frame-canaries when called with different reference-canaries and return correctly in both cases.

If the function that changed the reference-canary returns (time 8) there will a canary mismatch which will result in a process abort.



```
void bar(){
  return;
}
void qux(){
  bar();
  return;
}
void foo(){
  bar();
  renew_canary();
  qux();
  return;// Fails
}
int main(){
  foo();
  return 0;
}
```

Fig. 2. Stack evolution of a program which changes the reference-canary.

Example 2: Figure 3 shows the state of the stack and the reference-canary on a forking-server using the RAF SSP technique. All the stack frames used by a child have the same frame-canary value, which is different from the father and from other children. Since each child process defines an error confinement region, our strategy randomises the canary on each confinement region.

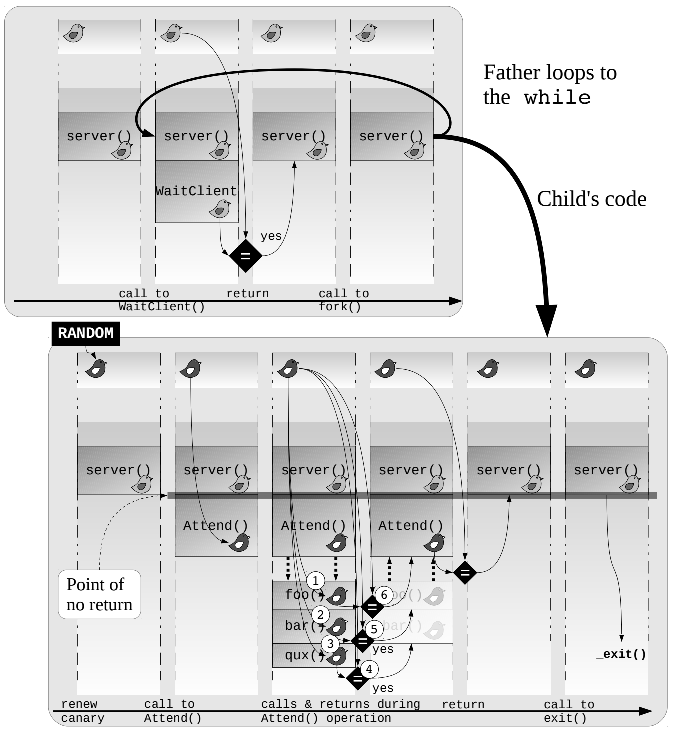


Fig. 3. Stack evolution of the code on listing 1.

In the case of a forking server, each client is attended by a different child process. Since the child always terminates after attending the client, the canary is renewed on every connexion regardless whether it is requested by a legal client or an attacker. On a pre-forked architecture, a child is running and attending requests until it is killed (by the main server process to reduce the number of active processes, or because it has been crashed as a result of an attack).

```
void server(){                    |   void Attend(){
  ...                             |     foo();
  while(1) {                      |     ....
    client=WaitClient();          |   }
    if (fork()==0) {              |   int foo(){ bar(); }
    renew_canary();               |
    Attend();                     |   int bar(){ qux(); }
    _exit();                      |
  }                               |   char qux(){};
}  ...                            |
```

Listing 1. Basic forking server example.

Taking into account observation 4, the frame-canary of a non-returning function is never checked against the reference-canary. Therefore a non-returning function can safely change the value of the reference-canary. All subsequent function calls (invoked from the non-returning function) will use the new reference-canary for building the stack frame. Upon returning, the canary checks will match because the reference-canary and the frame-canary will be the same (assuming the stack has not been smashed).

D. Special considerations

The `setjmp()/longjmp()` library provides control flow facility that breaks the "normal" control flow of functions: call and return sequence. The `setjmp()` saves the stack context/environment in a variable that can later be restored by `longjmp()` function. This backward jump does not involve any return operation and does not check the integrity of the frame-canaries, therefore it can be safely used. But, if the reference-canary has been renewed between the time when the `setjmp()` was called and when the `longjmp()` is invoked, then there will be a mismatch between the reference-canary and stack-canary on the function where the `setjmp()` was performed.

This special case can be addressed in two different ways:

- Do not return from functions that call `setjmp()`. This behaviour can be forced by declaring that functions as `noreturn`. Note that the function that uses `setjmp()` can call nested functions at any time, but can not return.

- A more robust and transparent to the application solution consist on modifying the struct where `setjmp()` saves the context/environment by adding a new field to store the value of the reference-canary at the time the `setjmp()` is called. Later the `longjmp()` will restore the reference-canary value using the stored one. With this solution there is no limitation of the control flow.

V. Implementation

A prove of concept of the of the proposal has been implemented as a shared library which overrides the `fork()` call. The library is called `libraf.so`, the code of the library is in listing 2.

The `fork()` function, on listing2, calls the native `fork()` function and then the reference-canary of the child is renewed calling the `renew_rnd_stack_chk_guard()` function, which is basically a copy of the code library code to setup the canary. All but one bytes are random values read from `/dev/urandom`, which is the same source of randomness that the one used by the standard canary.

```
#ifdef __i386__
# define THREAD_SET_STACK_GUARD(x) \
  asm ("mov %0, %%gs:0x14" ::"r" (x) : "memory");
#elif defined __x86_64__
# define THREAD_SET_STACK_GUARD(x) \
  asm ("mov %0, %%fs:0x28" ::"r" (x) : "memory");
#endif
pid_t (*native_fork) (void);
static void __raf_fork_init(void) {
  native_fork = dlsym(RTLD_NEXT, "fork");
  if (NULL == native_fork) {
    fprintf(stderr, "Error in `dlsym`: %s\n",
        dlerror());
  }
}
static void renew_rnd_stack_chk_guard(void) {
  union {
    uintptr_t num;
    unsigned char bytes[sizeof (uintptr_t)];
  } ret;
  const size_t ranb = sizeof(ret.bytes) - 1;
  ret.num = 0;
  int fd = __open("/dev/urandom", O_RDONLY);
  if (fd >= 0) {
    if (__read(fd, ret.bytes + 1, ranb) == ranb){
      THREAD_SET_STACK_GUARD(ret.num);
    }
    __close (fd);
  }
}
pid_t fork(void) {
   pid_t pid;
   if (native_fork==NULL) __raf_fork_init();
   pid = real_fork();
   if (pid == 0) renew_rnd_stack_chk_guard();
   return pid;
}
```

Listing 2. Implemented as a pre-load shared library: `libraf.so`.

Although the compiler will not add stack protector code to none of this functions because they do not have local buffers (larger than 8 bytes), it is advisable to compile the shared library with the option `-fno-stack-protector` to be sure that the compiler does not add it. Otherwise, the canary renewal will be detected as a stack corruption and the program aborted.

In order to use the new version of `fork()`, the server has to be launched with the `LD_PRELOAD=libraf.so` as follows:

```
$ LD_PRELOAD=libraf.so apachectl start
```

Another way to include the proposal in a running system is by modifying the standard C library. We tested the proposal modifying the code of the `fork()` function of the GNU eglibc. Nevertheless, for brevity we describe it as a new library call, `raf_fork()`. The modification basically consists on calling again the canary setup code `_dl_setup_stack_chk_guard`) right after calling fork.

```
void renew_canary(void) {
  /* Renew the stack checker's canary. */
  uintptr_t stack_chk_guard =
       _dl_setup_stack_chk_guard (NULL);
#ifdef THREAD_SET_STACK_GUARD
  THREAD_SET_STACK_GUARD (stack_chk_guard);
#else
  __stack_chk_guard = stack_chk_guard;
#endif
}
pid_t raf_fork (void) {
   if (fork()==0) renew_canary();
   return pid;
}
```

Listing 3. Implemented as a new service: `raf_fork()`.

The modification of the `setjmp/longjmp` family consist on increasing the size of the `jmp_buf` array, to store the value of the current reference-canary at the `setjmp` (which is done with just one assignment instruction), and restore it on the `longjmp` function.

VI. Statistical evaluation

We analyse the protection provided by RAF SSP as both, a stand-alone technique and when combined with the ASLR and NX techniques. The cost is measured as the number of attempts (trials) needed by the attackers to break-in the system. Since the NX technique prevents remote code injection, in order to exploit a stack smash vulnerability it is necessary to bypass the canary plus the ASLR at once, presented in section VI-B. The unrealistic attack to the canary as a stand alone technique has been included for completeness.

Let k be the number of trials until the system is defeated, let c the number of different values that can take the canary, and let j by the number of different positions where the ASLR can place the code.

The system is broken when the secrets are correctly guessed. We are interested in the probability distribution of the process defined as: "*the probability that the first success requires k number of trials*". Larger values of k is good for the defenders and bad for attackers.

The plots shown in figures 4 and 5 are for a standard Ubuntu 32 bits system, where the canary has 3 bytes (which gives a range of $c = 2^{24}$ values) and the ALSR has 8 bits (256 values) of entropy. Note that the 256 values of the ASLR entropy is the best case for the attackers assuming that only a mapped library (typically the libc) it is enough to build the attack.

A. Bypassing only the canary

As described in section III there are two main ways to attack the canary: full search and byte-for-byte.

Full search attack: Statistically, the brute force attack is described as a "**sampling without replacement**" and since all the values has the same probability ($\frac{1}{c}$) it is modelled by the uniform distribution with a support range of $[1, c]$ and a mean of $\frac{c+1}{2}$.

When the RAF SSP is used, the attacker can not do a brute force attack because incorrect values can not be discarded. The only strategy for an attacker is to select a valid canary value (it does not matter the value) and keep trying the same value until the canary matches with the randomly chosen by the server. The attacker can change the value of the tried canary but it does not improve the chances to have a match. This trial process is described as "**sampling with replacement**" and it is modelled by a geometric distribution with a support range of $[1, \infty[$ and the mean is c.

The standard SSP requires at most 2^{24} trials[1] to be sure that the canary value is found, but with the RAF SSP it is impossible to cover all the cases. On average, the RAF SSP requires only 2 times more trials to be broken. Three times the mean is needed to have a 95% probability of breaking in.

Byte-for-byte attack: On a byte-for-byte attack, the process of finding each byte is modelled as a uniform distribution whose mean is $256/2$ and the support range is $[1, 256]$. The attack to the three (for 32-bit systems) or seven bytes (64-bits systems) is modelled as the sum of 3, or 7, uniform random variables respectively. Using the central limit theorem the resulting distribution can be approximated to a Normal distribution with $\mu = 256n/2$, where n is the number of random bytes of the canary ($n = log_2(c)/8$), and support of $[1, 256n]$.

When the RAF SSP is used, the byte-for-byte attack can not be employed because any incorrect guess will trigger a renew of the full canary (all the bytes), which invalidates any previous correctly guessed byte. Therefore, the attacker is forced to use the sampling with replacement against the full canary. Figure 4 compares the cumulative distribution function (CDF) of the attack to a b-f-b exploitable overflow using the standard SSP and the proposed RAF SSP. The x-axis is in logarithmic scale.

With the standard SSP, the attacker needs at most 768 trails to break the system (and 384 in average). With this figures, the standard canary technique provides a weak protection for this kind of bugs. On the other hand, the RAF SSP disables the ability to split the attack into bytes, and so, the same attacker requires $2^{24} = 16 \times 10^6$ trials on average to break it (it is a geometric distribution with $\mu = c$), which represents an **improvement of 5 orders of magnitude**.

[1]On an 32bits Ubuntu.

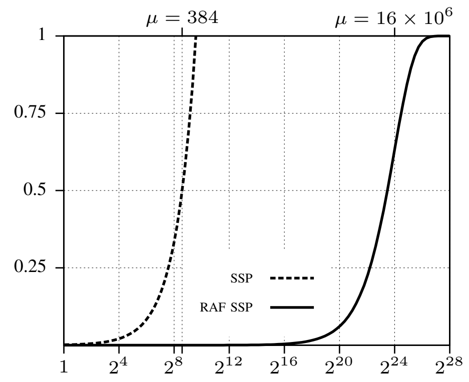


Fig. 4. Byte-for-byte vulnerability.

B. Bypassing SSP + ASLR + NX

In a real system, all these three complementary techniques are used simultaneously. Therefore, in order to exploit the fault, the attackers must:

1) Bypass the NX protection. The NX prevents the execution of injected code, the attackers are forced to "re-use" the code already present in the process by means of ROP programming.
2) Bypass the SSP protection. Which consists on finding the actual value of the canary.
3) Bypass the ASLR protection. Which requires to know the absolute address of the attacker's code, i.e. the entry point of the ROP sequence.

The attackers have to prepare/program the ROP sequence offline. If we suppose that the attackers know the code of the server then only the address of the entry point to the ROP sequence is unknown, the rest of the ROP sequence is relative to that entry point.

It is important to note that both, the ASLR and the canary techniques, have the same weakness on forking servers: all the children have the same memory layout as well as the same canary value. Unfortunately, as far as the authors know, there is not a simple method to re-randomise the ASLR on forked children. We will assume that all the forked children have the same ASLR value which allows the attacker to perform a brute force on it. The attacker has to create a brute force (or a probabilistic) attack to obtain the value of the two secrets: i) **canary value** and ii) **ROP entry address**. Let j be the number of different values where the ASLR algorithm can map the memory (i.e. the ASLR entropy).

We assume that the return address can only be overwritten if the canary is known. That is, the server's code checks always the canary first, and only returns from the function if it is correct. This behaviour allows the attacker to split the attack in two phases: first the value of the canary is found and then the value of the ROP entry address. A detailed analysis of how the three techniques can be bypassed can be found in [11]. The attack to the ASLR follows the same pattern as the attack to the canary, it is a uniform distribution, sampling without replacement, whose mean is $j/2$ and the support is $[1, j]$.

Using the standard SSP, the attack to the server is modelled as the sum of two uniforms distributions (which gives a trapezoidal distribution). If one of the uniforms has a much larger support range than the other ($c \gg j$), as it is in our case, the sum can be approximated by a simple uniform distribution with a mean of $(c + j)/2$ and a range of $[2, c + j - 1]$.

When the RAF SSP is used, it is not possible to split the attack because any incorrect guess (either canary value or ROP entry point) causes a canary renewal[2]. Since $c \gg j$, it is possible to approximate the resulting distribution to a geometric with a mean of $c \times j$ and support $[2, \infty[$.

Figure 5 shows the success probability of an attack on a standard system and when the RAF SSP technique is used on a real system. The x-axis is in logarithmic scale.

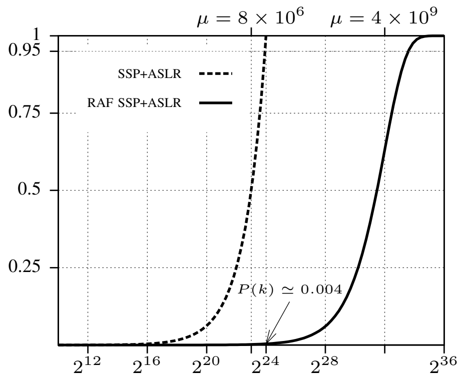


Fig. 5. Success of an attack against both: SSP and ALSR.

When the RAF SSP is used, the cost of the attack is calculated as result of the **multiplication** of the cost of each part: canary and ASLR. While in a normal system it is only the **sum** of each part. It takes at most $2^{24} + 2^8$ trials to break in a standard system, the system is broken with a 100% of probability. Using the RAF SSP technique, the chances of breaking in the system using the same number of trials is only 0.004%. On average, a 32bits system using the RAF SSP **requires** 512 **times more trials** to break.

VII. Experimental evaluation

The RAF SSP relies on the infrastructure of the standard SSP. Therefore, it does not increases the runtime cost operation of the application except when the fork operation is invoked. The temporal overhead can be reduced to the cost of generating a random word every time the `fork()` system call is called.

The **temporal cost** of renewing the canary is determined by the *r*enew_randomly_stack_chk_guard() function. The average cost of calling this function one million times is $2\mu s$ on an Intel Core™ 2 Duo CPU at 2.4Ghz.

Regarding the **spatial cost**: the memory size of the application is not increased, no new data structures are used, and no new code is generated by the compiler; also, no new data structures or buffers are needed in the library, only a few lines of code have to be included.

The RAF SSP has been tested with the following network servers: apache2, lighttpd, proftpd and samba. We will describe only the apache2 benchmarks.

[2]A detailed analysis of attacker strategies and the associated statistical distribution is beyond the scope of this paper. It will be published in an incoming paper.

We have used the Apache (apache2-mpm-prefork) binary included in the Ubuntu (12.04) distribution with the default configurations 4 and the *Apache HTTP server benchmarking tool (ab)* tool to generate the client workload.

```
<IfModule mpm_prefork_module>
   StartServers      5
   MinSpareServers   5
   MaxSpareServers  10
   MaxClients      150
   MaxRequestsPerChild 0
</IfModule>
```

Listing 4. Apache configuration parameters.

The ab tool was configured to perform requests of size 1KB, 10KB and 100KB, and each size was tested with 10, 50 and 100 concurrent requests (concurrency column). Each experiments consists of 10^6 requests.

		Latency (ms)		**Throughput (KB/s)**	
	Concurrent	SSP	RAF SSP	SSP	RAF SSP
1KB	10	0.094	0.093	12	12
	50	0.097	0.097	12	12
	100	0.097	0.098	12	12
10KB	10	0.095	0.095	104	104
	50	0.099	0.099	101	100
	100	0.102	0.101	98	99
100KB	10	0.135	0.135	725	723
	50	0.142	0.143	690	683
	100	0.164	0.164	598	596

TABLE II. Performance overhead comparative.

Table II shows the average of each experiment. There is not significant differences between standard and RAF SSP apart from the variability introduced by the processor and operating system features. The small overhead caused by RAF SSP is practically undetectable when analysing the complete operation of the server.

We have installed a modified version of the eglibc (with the RAF SSP enabled at fork) in a Ubuntu Linux distribution, and all the tested applications worked correctly: all graphical services, several browsers, several text editors (LibreOffice), java Open JDK interpreter, etc.

VIII. Discussion

The sequence `fork() + exec()` provides a very robust security schema because of the strong decoupling between the father and the children. The exec call renews both the value of the canary and the addresses of the code (ASLR). With respect to the canary, our technique is as powerful as calling exec but without the high overhead of the exec call.

Contrarily to other approaches where the canary is different in each stack frame [12], our approach changes the canary for every error confinement region. From the attacker point of view, the target stack frame is the one that belongs to the faulty function. The rest of the stack frames have no interest to the attacker. On a network server, only the code executed by the child sever are prone to attacks and in that code the canary is always re-randomised by the RAF SSP.

	Standard SSP		RAF SSP		
	Trials to break in 100%	Mean	Trials to break in 100%	Mean	Mean increased by a factor of
SSP_bfb	3×2^8	3×2^7	∞	2^{24}	43691
SSP_full	2^{24}	2^{23}	∞	2^{24}	2
SSP_bfb+ASLR	$3 \times 2^8 + 2^8$	2^9	∞	2^{32}	32768
SSP_full+ALSR	$2^{24} + 2^8$	$2^{23} + 2^7$	∞	2^{32}	512

TABLE I. STANDARD SSP VERSUS RAF SSP ($c = 2^{24}, j = 2^8$).

Our technique does not modify the coverage protection of the SSP technique. If a programming error can be exploited in such a way that the SSP technique fail to detect it, then RAF SSP will fail too. But if a stack smashing is detected, then our modified system will also detect it and makes more difficult to exploit it.

Since the overhead of the RAF SSP occurs only at the fork call, the less forks are done by the server the less overhead. For example, a pre-forked server have zero overhead as far a no new children are launched. If the server children processes are being killed during an attack (due to incorrect guesses on brute force attack) then the new children have newly random canaries. That is, the RAF SSP acts only when it is effectively required while during normal operation it has zero impact.

In the sectionVI-B we evaluate the cost of an attack when the three most common protection techniques are simultaneously employed, and show that the combined effectiveness can be computed as the product of each one because the attacks can not be split. Since the canary is typically the first barrier to be bypassed, the RAF SSP causes the same multiplicative effect when combined with other techniques, as far as any incorrect trial on any technique applied afterwards will renew the canary and so the attacker will be forced to start over again.

IX. CONCLUSIONS

This paper reconsiders the idea of renewing the canary at ever new process creation (at fork time), and not only when a new image is loaded (at exec time). The current implementation of stack guard generates a new random canary for every new process image, that is, when the `exec()` is called. All the children processes inherit the value of the canary from the father. We propose to renew the canary value on every child. This is specially effective on multi-process networked servers where the main server forks processes to concurrently attend several clients.

We believe that a non accurate statistical understanding of the re-randomisation effect may had discouraged other authors from considering the benefits of the this technique. We show that re-randomise the canary improves the protection against attacks several, specially when combined with other commonly used protection techniques, several orders of magnitude with a negligible cost.

The new technique is called RAF SSP (Renew After Fork Stack Smashing Protector) and has the following properties:

- The RAF SSP strategy has a negligible overhead.
- The canary brute force attack, specially the byte-for-byte variant, is not longer possible.
- While the attack to the standard SSP follows a Uniform distribution, the attack to the RAF SSP is a Geometric distribution.
- The solution can be implemented by means of a preloaded share library, therefore it does not require to modify the source code of the server or recompile it, nor modify the operating system, neither the system libraries.
- The RAF SSP has been validated with several network servers: apache2, lighttpd, proftpd and samba without modifying the source code nor recompiling.

ACKNOWLEDGMENT

Work supported by Spanish MICINN grant TIN2009-14205-C04-02 and MEC TIN2012-38308-C02-01.

REFERENCES

[1] Y. Younan, D. Pozza, F. Piessens, and W. Joosen, "Extended protection against stack smashing attacks without performance loss," in *In Proceedings of ACSAC*, 2006.

[2] A. One, "Smashing the stack for fun and profit," *Phrack*, vol. 7, no. 49, 1996.

[3] Bulba and Kil3r, "Bypassing stackguard and stackshield," *Phrack*, 56, 2002.

[4] G. Richarte, "Four different tricks to bypass stackshield and stackguard protection," *World Wide Web*, vol. 1, 2002.

[5] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh, "On the effectiveness of address-space randomization," in *Proceedings of the 11th ACM conference on Computer and communications security*, ser. CCS '04. New York, NY, USA: ACM, 2004, pp. 298–307. [Online]. Available: http://doi.acm.org/10.1145/1030083.1030124

[6] M. Tran, M. Etheridge, T. Bletsch, X. Jiang, V. Freeh, and P. Ning, "On the expressiveness of return-into-libc attacks," in *Proceedings of the 14th international conference on Recent Advances in Intrusion Detection*, ser. RAID'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 121–141. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-23644-0_7

[7] B. Erb, "Concurrent programming for scalable web architectures," Diploma Thesis, Institute of Distributed Systems, Ulm University, April 2012. [Online]. Available: http://www.benjamin-erb.de/thesis

[8] C. Cowan, C. Pu, D. Maier, H. Hintongif, 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 Symposium*, Jan 1998, pp. 63–78.

[9] 'xorl'. (2010) Linux GLibC Stack Canary Values. [Online]. Available: http://xorl.wordpress.com/2010/10/14/linux-glibc-stack-canary-values/

[10] H. Etoh, "GCC extension for protecting applications from stack-smashing attacks (ProPolice)," 2003. [Online]. Available: http://www.trl.ibm.com/projects/security/ssp/

[11] A. 'pi3' Zabrocki, "Scraps of notes on remote stack overflow exploitation," November 2010. [Online]. Available: http://www.phrack.org/issues.html?issue=67&id=13#article

[12] Y.-J. Park and G. Lee, "Repairing return address stack for buffer overflow protection," in *Proceedings of the 1st conference on Computing frontiers*, ser. CF '04. New York, NY, USA: ACM, 2004, pp. 335–342. [Online]. Available: http://doi.acm.org/10.1145/977091.977139