Protections

20200203

Outline

How to protect against vulnerabilites

Stack canaries

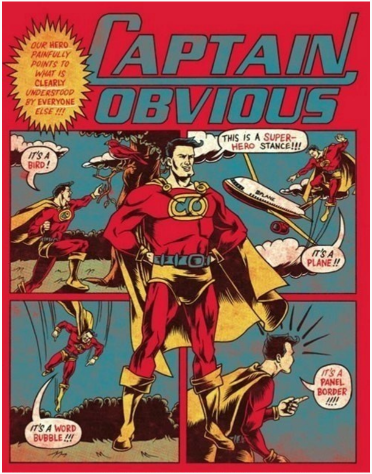
Executable space protection

ASLR

CFI on execution

How to protect against vulnerabilites

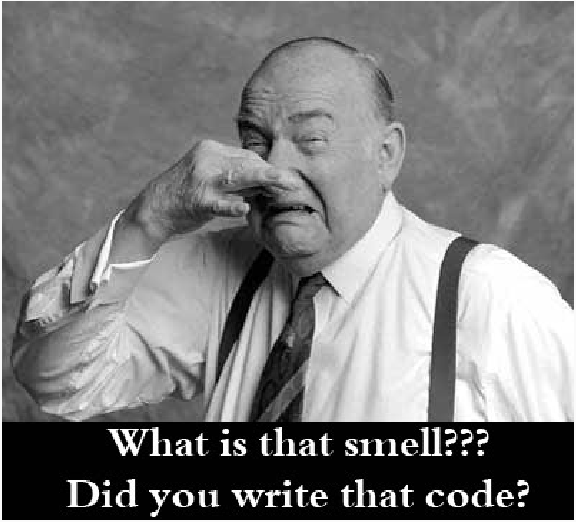
Write correct code, obviously . . .



Some people write fragile code and some people write very structurally sound code, and this is a condition of people.
– K. Thompson

To err is human, but to really foul up requires a computer.
– Anon

Use help/mitigation against bad code



Stack canaries

Stack canaries

What it is

A public canary value is placed right above function-local stack buffers in the stack frame.

Its integrity is checked prior to function return.

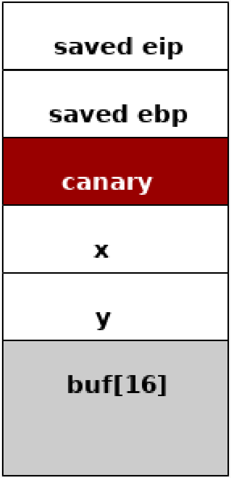
AKA cookie, stack cookie

What it provides

Ensure the saved base pointer and function return address have not been corrupted

Needs compiler support only

How it looks



Summary

The good ✔

- Pure compiler-based solution (no OS support)
- Most stack-based buffer overflows are countered

The bad ✖

- Protect only variables above it in the stack
- Not always active
- Sometimes the cookie can be guessed (see later)

Implementations

VS `/Gs[size]`

If a function requires more than size bytes of stack space for local variables, its stack probe is initiated. By default, the compiler generates code that initiates a stack probe when a function requires more than one page of stack space (i.e. `/Gs4096`).

GCC `-fstack-protector`

Emit extra code to check for buffer overflows, such as stack smashing attacks. This is done by adding a guard variable to functions with vulnerable objects. This includes functions that call alloca, and functions with buffers larger than 8 bytes.

Terminator canary

Definition

A terminator canary is comprised of common termination symbols, such as '\0' (0x00), ” (0x0a), ’˚ (0x0d), EOF (-1)

Example: `0x000a0dff`

Effectiveness

The attacker cannot use common C string libraries, since copying functions will terminate on the termination symbols.

- Either the attack is detected (canary does not hold the same value)
- Or it stops it due to termination symbols.

Random canary

Definition

The loader chooses a word-sized (32/64 bits) random canary string on program start.

Effectiveness

The randomness makes the value of the canary hard to guess

Behavior

```c
#include <string.h>

int main(int argc, char *argv[])
{
    char buf[10];
    strcpy(buf, argv[1]);
    return buf[5];
}
```

StackGuard effectiveness (Cowan et al., 2000)

Program	without protection	with protection
dip 3.3.7	root shell	program halts
elm 2.4	root shell	program halts
perle 5.003	root shell	program halts
Samba	root shell	program halts
SuperProbe	root shell	program halts
umount / libc 5.3.12	root shell	program halts
wwwcount 2.3	httpd shell	program halts
zgv 2.7	root shell	program halts

Considerations

Efficiency

Canary checks for every function causes a performance penalty.

$\approx$ 8% for Apache

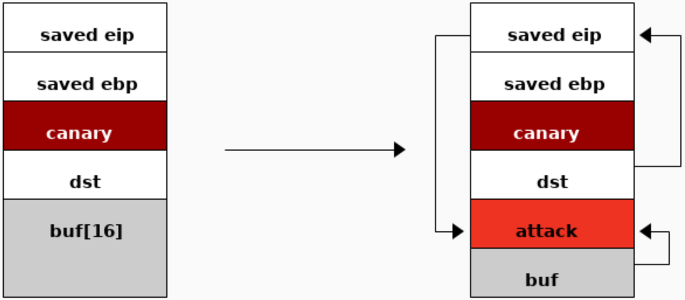
PointGuard

Canaries are also placed next to

- function pointers
- `setjmp` buffers

Greater performance impact

Defeating canaries



Example vulnerable on prior versions

```c
int f (char ** argv)
{
        int pipa;        // useless variable
        char *p;
        char a[30];

        p=a;

        printf ("p=%x\t -- before 1st strcpy\n",p);
        strcpy(p,argv[1]);        // <== vulnerable strcpy()
        printf ("p=%x\t -- after 1st  strcpy\n",p);
        strncpy(p,argv[2],16);
        printf("After second strcpy ;)\n");
}

int main (int argc, char ** argv) {
        f(argv);
        execl("back_to_vul","",0);  //<-- The exec that fails
        printf("End of program\n");
}
```

Weakness of canary randomization

Canary is randomized whenever `libc` is loaded.

That is every time, `execve()` is used . . .

but not when `fork()` is used

Brute-forcing the canary

Technique :: Byte-per-byte brute-forcing

- On average ≈ 512 attempts
- Brute-force + timing analysis
- Incorrect guesses fail fast, correct guesses fail slow

Limitations

- Need the canary to stay the same (i.e. forking daemons)

Canaries for every one

```c
#include <stdio.h>

/* Commenting out or not using the string.h header will cause this
 * program to use the unprotected strcpy function.
 */
#include <string.h>

int main(int argc, char **argv)
{
	char buffer[5];
	printf ("Buffer Contains: %s , Size Of Buffer is %d\n",
		buffer,sizeof(buffer));
	strcpy(buffer,argv[1]);
	printf ("Buffer Contains: %s , Size Of Buffer is %d\n",
		buffer,sizeof(buffer));
}
```

In a nutshell

Performance	- several instructions per function - a few % - removable in safe functions
Deployment	No code change / recompilation
Compatiblity	100%
Safety guarantee	None

Executable space protection

Broad idea

- C does not specify what happens when a data pointer is used as if it were a function pointer (implementation-defined)
- Self-modifying code is pretty rare — outside of efficient JIT compilers

Idea

- Mark data memory as non-executable
- Needs OS support

Implementations

OS	Date	Version	Name(s)
OpenBSD	2003	3.3	`W^X`
Windows	2004	XP	DEP
FreeBSD	2004	5.3	
Linux	2004	2.6	PaX, ExecShield
macOS	2005	10.4	
macOS	2007	> 10.5	

Implementation details

NX/XD/XN bit

Modern AMD/Intel/ARM machines have a dedicated bit which flags memory pages as writable or else executable.

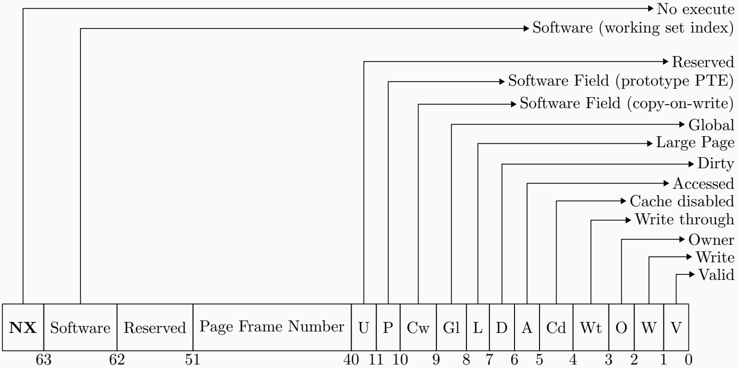
When set, the page is not executable

x86's original 32-bits table did not have such a mechanism.

Other implementations

- On x86, the mechanism is sometimes emulated (through CS segment)
- PaX NX also emulates the functionality on 32-bits

In (excruciating) details



Limitations

Warning

Data Execution Prevention does nothing to prevent a buffer overflow to rewrite the saved frame pointer or the saved instruction pointer (aka. return address).

A single call to `SqlExe("drop table ...")` is thus manageable.

Counterattacks

- Indirect code injection
- Jump-to-`libc` attacks
- Data-only attacks

Return-oriented programming

Definition

Return oriented programming (ROP) is an exploit technique

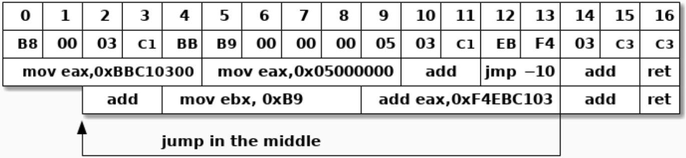
1. Gains control of the call stack
2. Executes carefully chose machine instruction sequences already present called gadgets

Remarks

- There exist Turing-complete sets of gadgets
- This is an extension to `return-into-libc` attacks

Overlapping instructions (J. Kinder)

Other instructions are embedded inside your instructions.



This can be used to find gadgets inside your code, e.g. `jmp esp` (`0xffe4`)

Gadgets

- Gadgets ending with a ret are typically found in function epilogues
- Tools (ropper, ROPgadget, ...) help in finding gadgets and ROP chains to

Origin

- Intended instructions
- Unaligned bytes

Build

- String gadgets into units of functionality (loads/stores, jumps, arithmetic)
- Goal : execute another shellcode

Basic example

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

void not_called(int pseudo_arg)
{
     printf("Enjoy your shell!\n");
     system("/bin/bash");
}

void vulnerable_function(char* string)
{
     char buffer[100];
     strcpy(buffer, string);
}

int main(int argc, char** argv)
{
    vulnerable_function(argv[1]);
    return 0;
}
```

More involved example

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

char* not_used = "/bin/sh";

void not_called(int pseudo_arg) {
    printf("Not quite a shell...\n");
    system("/bin/date");
}

void vulnerable_function(char* string) {
    char buffer[100];
    strcpy(buffer, string);
}

int main(int argc, char** argv) {
    vulnerable_function(argv[1]);
    return 0;
}
```

kBouncer (Pappas et al., 2013)

Observation 1

- ROP attacks issue returns to *non-call-preceded* addresses
- Make all return instructions target call-preceded addresses

Observation 2

- ROP attacks are built of long sequences of short gadgets
- Do not allow long sequences of short gadgets

Based on stack history, decide to abort

Anti-ROP

State-of-the-art

Lightweight ROP countermeasures are still exploitable

Stronger defenses

- G-Free (K. Onarlioglu et al. 2010) remove unintended return instructions and encrypt return addresses

In a nutshell

Performance	no impact if hardware support <1% in PaX
Deployment	Kernel support (common) Modules opt-in
Compatiblity	Can break JIT compilers, unpackers
Safety guarantee	Code injected to NX page Never eXecutes *but one does not need it . . .*

ASLR

Address-space Layout Randomization

Definition

ASLR is a technique to prevent exploitation of memory corruption vulnerabilities.

It rearranges the address space positions of a process, e.g., the base of the executable, the stack, the heap, and libraries.

Limitations

- Needs OS support
- ASLR + NX needs PIE

How it works

Most everything can be randomized that way :

- code
- global variables
- heap allocations, . . .

ASLR basically consists of randomly distributing the fundamental parts of a process (executable base, stack pointers, libraries, . . .)

Is it enabled ?

```
ldd /bin/ls
```

linux-vdso.so.1 (0x00007ffef19f7000)
libcap.so.2 => /usr/lib/libcap.so.2 (0x00007f6b2c68f000)
libc.so.6 => /usr/lib/libc.so.6 (0x00007f6b2c2d8000)
/lib64/ld-linux-x86-64.so.2 => /usr/lib64/ld-linux-x86-64.so.2 (0x00007f6b2cab5000)

```
ldd /bin/ls
```

linux-vdso.so.1 (0x00007ffdf0bad000)
libcap.so.2 => /usr/lib/libcap.so.2 (0x00007f5d4548f000)
libc.so.6 => /usr/lib/libc.so.6 (0x00007f5d450d8000)
/lib64/ld-linux-x86-64.so.2 => /usr/lib64/ld-linux-x86-64.so.2 (0x00007f5d458b5000)

What is actually randomized ?

```
cat /proc/self/maps | grep -E 'stack|heap|libc'
```

- Run 1

5608b9a39000-5608b9a5a000	rw-p	0	00:00	0	[heap]
7f1be2370000-7f1be251e000	r-xp	0	fe:02	2885816	/usr/lib/libc-2.26.so
7f1be251e000-7f1be271d000	—p	001ae000	fe:02	2885816	/usr/lib/libc-2.26.so
7f1be271d000-7f1be2721000	r–p	001ad000	fe:02	2885816	/usr/lib/libc-2.26.so
7f1be2721000-7f1be2723000	rw-p	001b1000	fe:02	2885816	/usr/lib/libc-2.26.so
7ffc28a7e000-7ffc28a9f000	rw-p	0	00:00	0	[stack]

- Run 2

56012b646000-56012b667000	rw-p	0	00:00	0	[heap]
7f080d001000-7f080d1af000	r-xp	0	fe:02	2885816	/usr/lib/libc-2.26.so
7f080d1af000-7f080d3ae000	—p	001ae000	fe:02	2885816	/usr/lib/libc-2.26.so
7f080d3ae000-7f080d3b2000	r–p	001ad000	fe:02	2885816	/usr/lib/libc-2.26.so
7f080d3b2000-7f080d3b4000	rw-p	001b1000	fe:02	2885816	/usr/lib/libc-2.26.so
7ffdab447000-7ffdab468000	rw-p	0	00:00	0	[stack]

Implementations

OS	Date	Version
OpenBSD	2003	3.3
Linux	2005	2.6.12
Windows	2007	Vista
macOS	2007	> 10.5

- FreeBSD still has no support in `-CURRENT`

Impact on execution

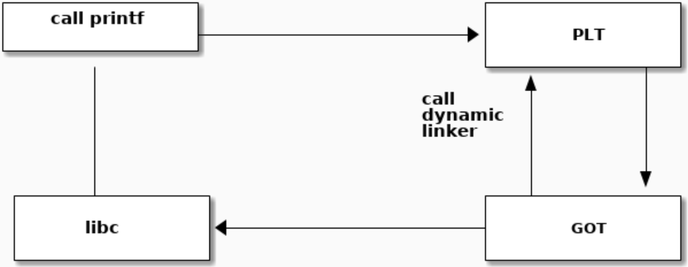
ASLR has a moderate impact ($\approx 3\%$) on performance

Attacking ASLR

- Parts of addresses are not randomized (i.e. GOT)
- Data and BSS segments are mapped to static locations. *Most applications have at least one interesting global*
- Any info leak disclosing location can be used to "guess" the where gadgets are.

.got & .plt

- GOT : Global Offset Table
- PLT : Procedure Linking Table



Further protections: RELRO

Definition

RELRO is a generic mitigation technique to harden the data sections of an ELF binary/process.

Partial RELRO

- `gcc -Wl,-z,relro`
- Reorders the binary : `.got`, `.dtors` precede data sections
- non-PLT GOT is RO
- GOT still writable

Full RELRO

- `gcc -Wl,-z,relro,-z,now`
- Partial RELRO + GOT is read-only

KASLR

Definition

KASLR randomizes the kernel code location in memory on system boot

Weakness

Memory distribution of kernel is unchanged once installed.

$\Rightarrow$ On next system restart no new random memory distribution will be performed.

Implementation

NetBSD	2017	current

KARL

Definition (OpenBSD)

Kernel binary files are generated by distributing the kernel's internal files in a random order each time the system is restarted or updated, so each system will work every time it is booted with a unique kernel totally different from other systems at binary level

Why KARL ?

Our immune systems work better when they are unique. Otherwise one airline passenger from Singapore with a new flu could wipe out Europe (they should fly to Washington instead).
Our computers should be more immune.
– Theo de Raadt

In a nutshell

Performance	Randomize once at load time
Deployment	Kernel support
	No recompilation needed
Compatiblity	Transparent to PIE programs
Safety guarantee	Not much in `x86`, better in `amd64`
	but one does not need code injection ...

CFI on execution

General idea

Compiler generates a static over-approximation of licit jump sites for all dynamic jumps.

At runtime, it is checked that jump targets are authorized.

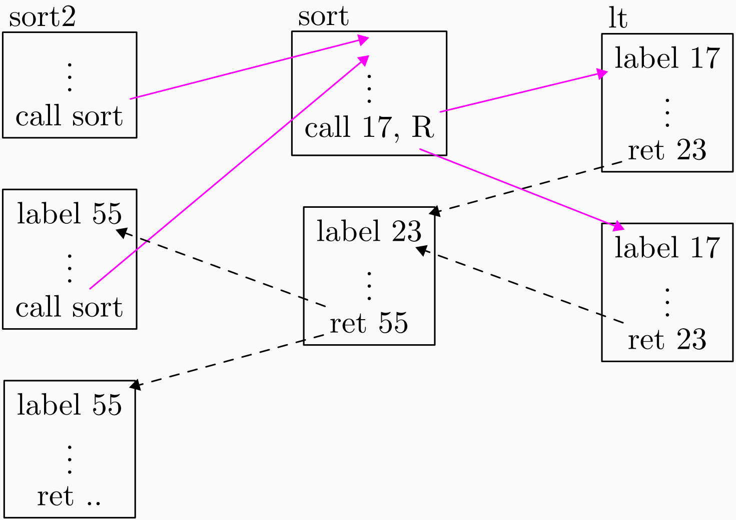
Example (U. Erlingsson et al.)

```
bool lt(int x, int y)
{
     return x < y;
}

bool gt(int x, int y)
{
     return x > y;
}

sort2(int a[], int b[], int len)
{
     sort(a, len, lt);
     sort(b, len, gt);
}
```

CFI enforcement



Property

The CFI security policy dictates that software execution must follow a CFG path determined ahead of time.

The CFI security policy needs be conservative: i.e. all valid executions should be allowed event at the cost of allowing invalid executions.

Overhead and slowdown

Code-size increase

$\approx$ 8%

Execution slowdown

0%–45% (mean: 16%)

Lightweight CFI

Control-flow destinations must be aligned on multi-word boundaries.

- Allow all basic blocks
- Basically only disallows jumping into overlapping instructions

Other measures

Sanitizers are runtime checkers dedicated to specific bugs

Memory sanitization (ASan)

Detect out-of-bound and use-after-free bugs

Undefined behavior sanitization (UBSan)

Detects the used of undefined behaviors at runtime

Impact

- 73% processing time
- 340% memory usage

Pre-summary

Protection	Exploitation
NX	easy
ASLR	feasible
canaries	depends
NX + ASLR	feasible
NX + canaries	depends
ASLR + canaries	hard
All 3	hard

Summary

Memory corruption vulnerabilities are well-addressed by the combination of

- `W^X`
- Stack canaries
- ASLR

Using only one of these techniques is not enough.

Compilers are including more advanced measures (CFI, sanitizers) to further mitigate these issues.

Questions ?

https://rbonichon.github.io/teaching/2020/asi36/