



Mitigating Overflows using Defense-in-Depth

What can your compiler do for you?

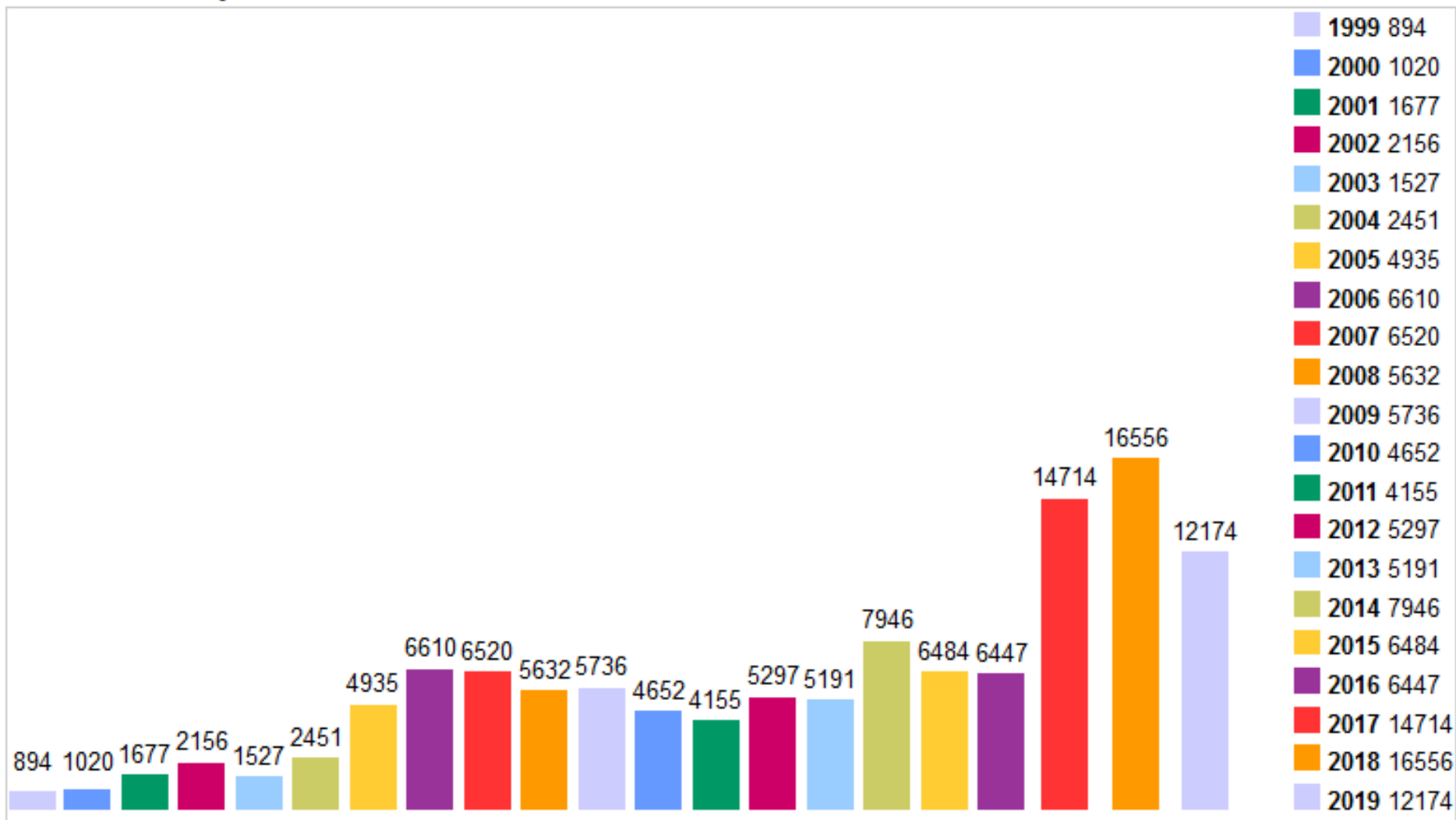
Who am I?



- ⊙ Javier Tallón – @javiertallon
- ⊙ More than 12 years working in IT Security
- ⊙ Full-stack ~~hacker~~ wannabe
- ⊙ Common Criteria Expert
- ⊙ Also PCI-PTS, FIPS 140-2, ISO 27K1, SOC2...
- ⊙ Cyber Security Teacher at UGR
- ⊙ CISSP Certified
- ⊙ jtsec Beyond IT Security - @jtsecES
- ⊙ Services
 - ⊙ Lightweight ITSEF (LINCE)
 - ⊙ Certification Consultancy
 - ⊙ Ethical hacking
- ⊙ Created in July'17 ~ 12 Employees
- ⊙ Based in Spain (Granada and Madrid)
- ⊙ careers@jtsec.es



Vulnerabilities By Year



- Is possible to program in a correct way?
- Is possible to demonstrate that a program is free of bugs?

Code breakthrough delivers safer computing

September 25, 2009

(PhysOrg.com) -- Computer researchers at UNSW and NICTA have achieved a breakthrough in software which will deliver significant increases in security and reliability and has the potential to be a major commercialisation success.

Professor Gernot Heiser, the John Lions Chair in Computer Science in the School of Computer Science and Engineering and a senior principal researcher with NICTA, said for the first time a team had been able to prove with mathematical rigour that an operating-system kernel - the code at the heart of any computer or microprocessor - was 100 per cent bug-free and therefore immune to crashes and failures.

The breakthrough has major implications for improving the reliability of critical systems such as medical machinery, military systems and aircraft, where failure due to a software error could have disastrous results.

"A rule of thumb is that reasonably engineered software has about 10 bugs per thousand lines of code, with really high quality software you can get that down to maybe one or three bugs per thousand lines of code," Professor Heiser said.

"That can mean there are a lot of bugs in a system. What we've shown is that it's possible to make the lowest level, the most critical, and in a way the most dangerous part of the system provably fault free."

"I think that's not an exaggeration to say that really opens up a completely new world with respect to building new systems that are highly trustworthy, highly secure and safe."

Verifying the kernel - known as the seL4 microkernel - involved mathematically proving the correctness of about 7,500 lines of computer code in a project taking an average of six people more than five years.

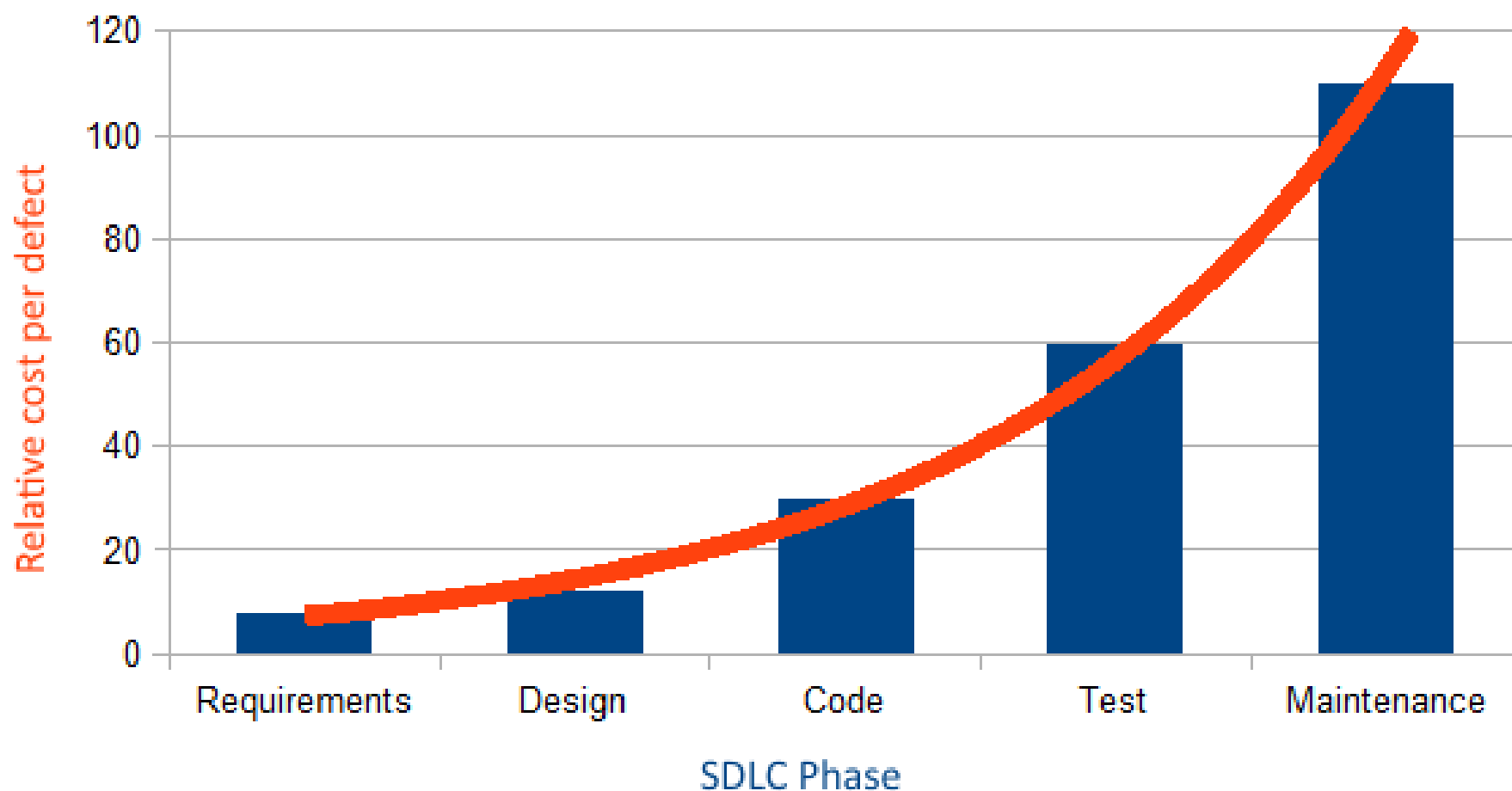
"The NICTA team has achieved a landmark result which will be a game changer for security-and-safety-critical software," Professor Heiser said.

"The verification provides conclusive evidence that bug-free software is possible, and in the future, nothing less should be considered acceptable where critical assets are at stake."

$(7500 / 6) / 5 =$
250 lines/man/year

Windows has about 50
mills lines of code

Cost of fixing software bugs



Cost of fixing software bugs

- ❑ What if it is a security bug?
 - ❑ Reputation
 - ❑ Loss of sales
 - ❑ Damage to the client (responsibilities)
 - ❑ Stock exchange market fall

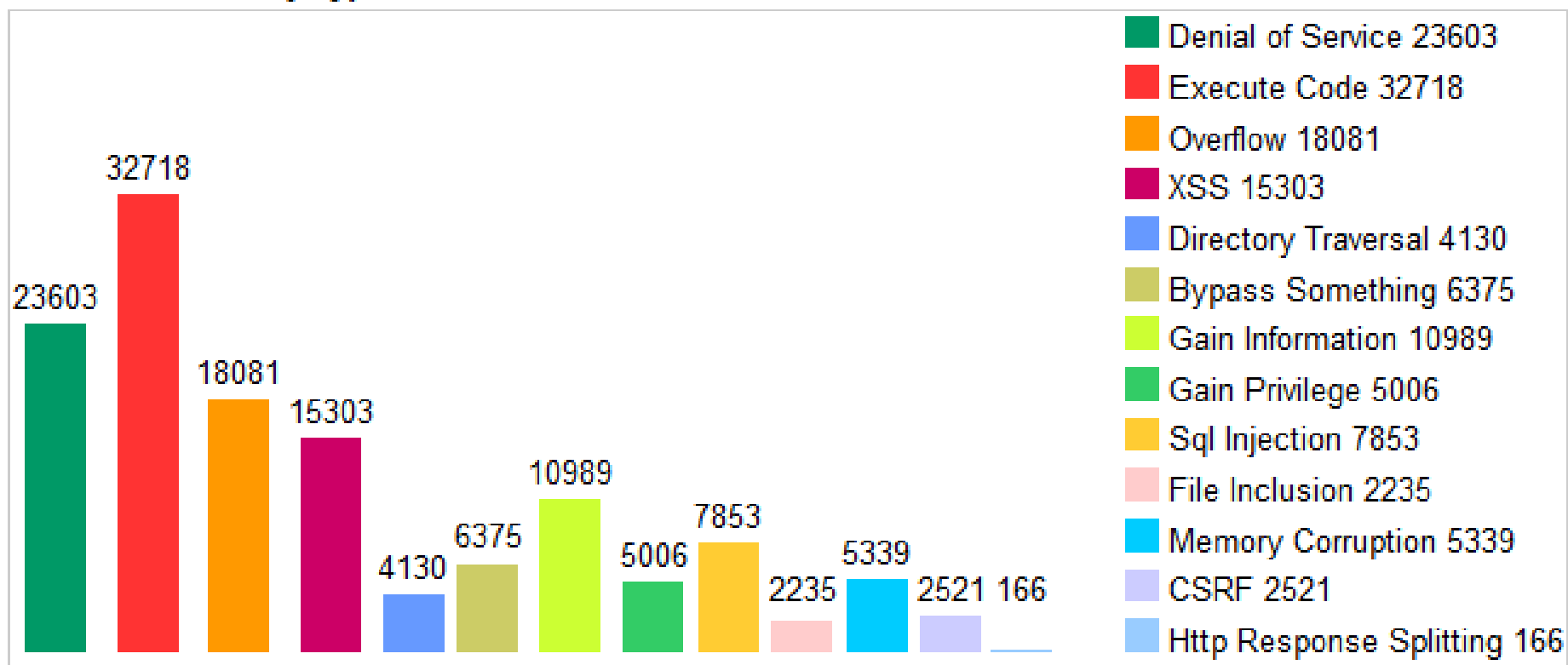


Difference between normal bug and security bug

- Users don't "have fun" looking for functional bugs
- A complex bug (to exercise) causes problems only to the user who finds it
- Programmers are committing less and less functional bugs
- Emphasis is on what the app should do (mostly positive requirements)
- Can be tested more easily

Buffer Overflows

Vulnerabilities By Type



Buffer Overflows

```
#include <string.h>
void foo (char *bar) {
    char c[12];
    strcpy(c, bar); // no bounds checking...
}

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



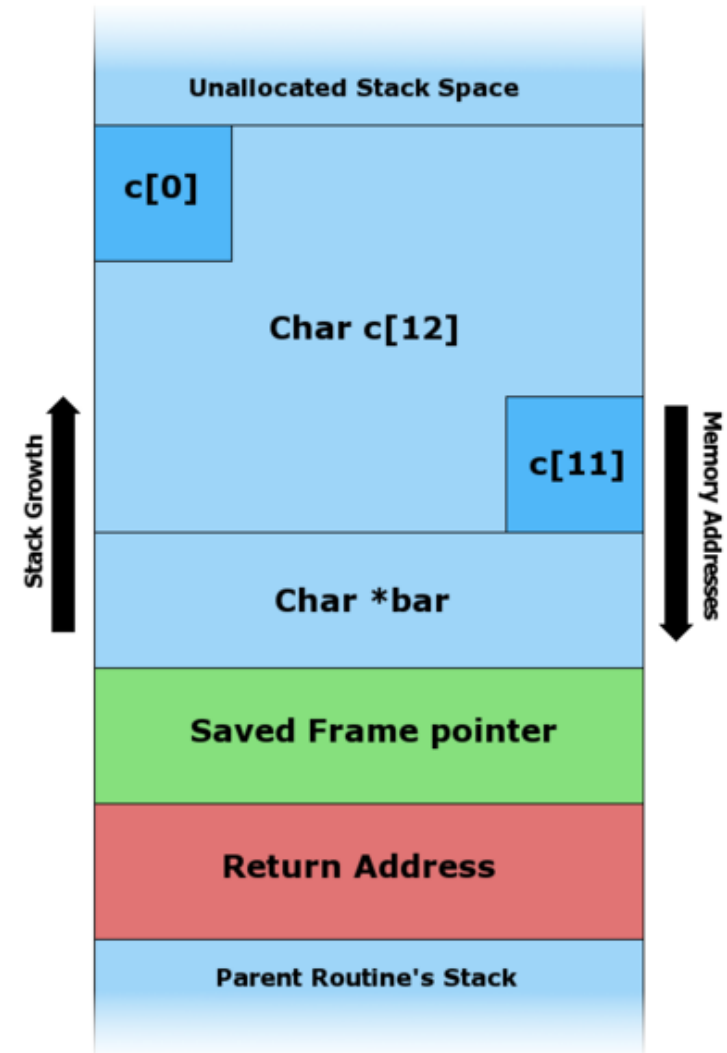
Buffer Overflows

```

#include <string.h>
void foo (char *bar) {
    char c[12];
    strcpy(c, bar); // no bounds checking...
}

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

```



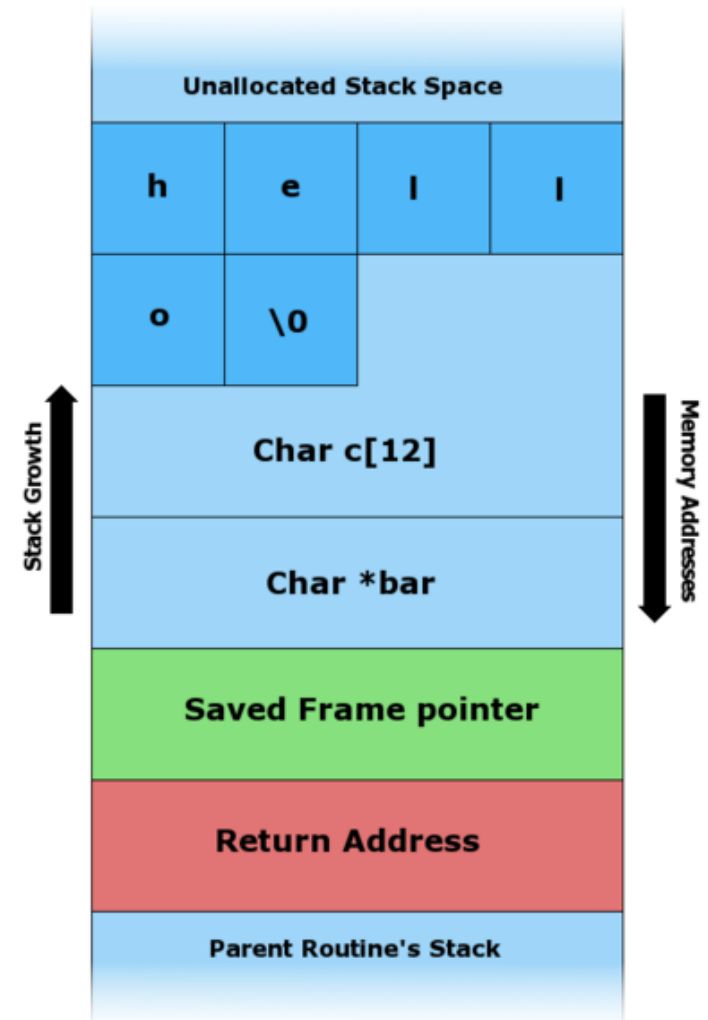
Buffer Overflows

```

#include <string.h>
void foo (char *bar) {
    char c[12];
    strcpy(c, bar); // no bounds checking...
}

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

```



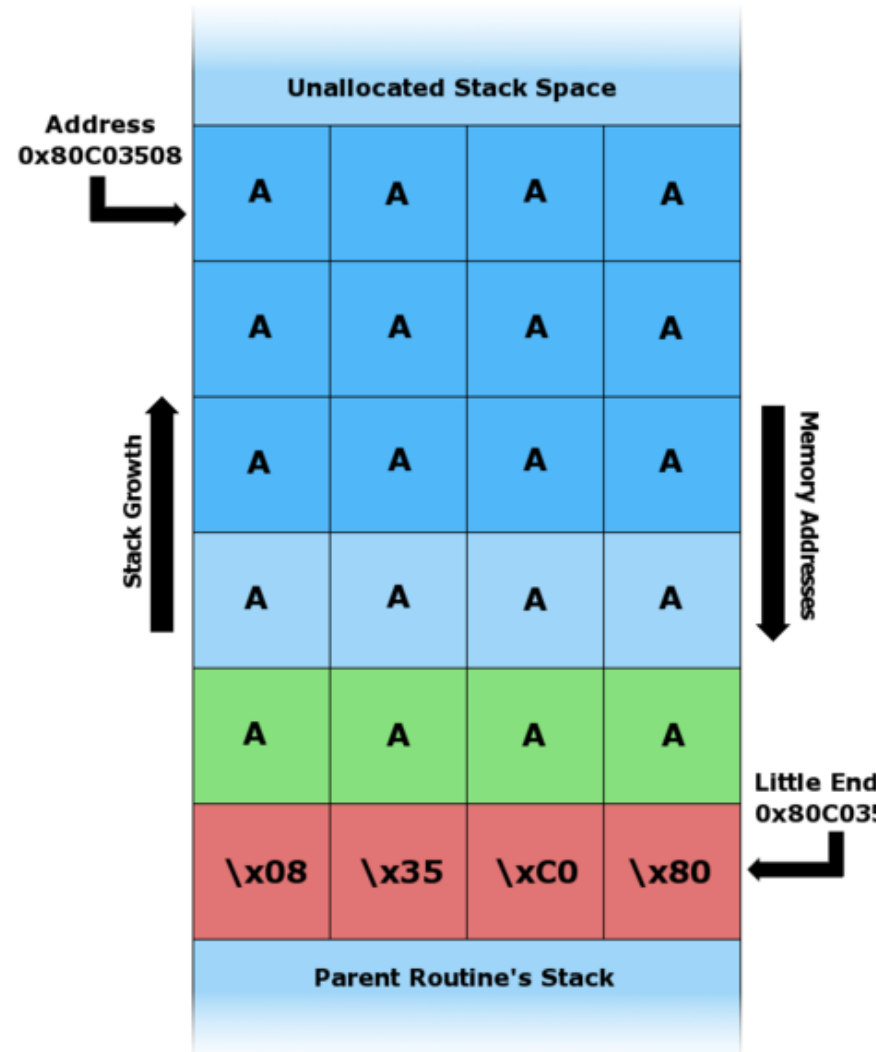
Buffer Overflows

```

#include <string.h>
void foo (char *bar) {
    char c[12];
    strcpy(c, bar); // no bounds checking...
}

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

```



Buffer Overflows

Dangerous C system calls

Extreme risk

- `gets`

High risk

- `strcpy`
- `strcat`
- `sprintf`
- `scanf`
- `sscanf`
- `fscanf`
- `vfscanf`
- `vsscanf`
- `streadd`
- `strecpy`
- `strtrns`
- `realpath`
- `syslog`
- `getenv`
- `getopt`
- `getopt_long`
- `getpass`

Moderate risk





- `getchar`
- `fgetc`
- `getc`
- `read`
- `bcopy`

Low risk

- `fgets`
- `memcpy`
- `snprintf`
- `strncpy`
- `strcadd`
- `strncpy`
- `strncat`
- `vsnprintf`

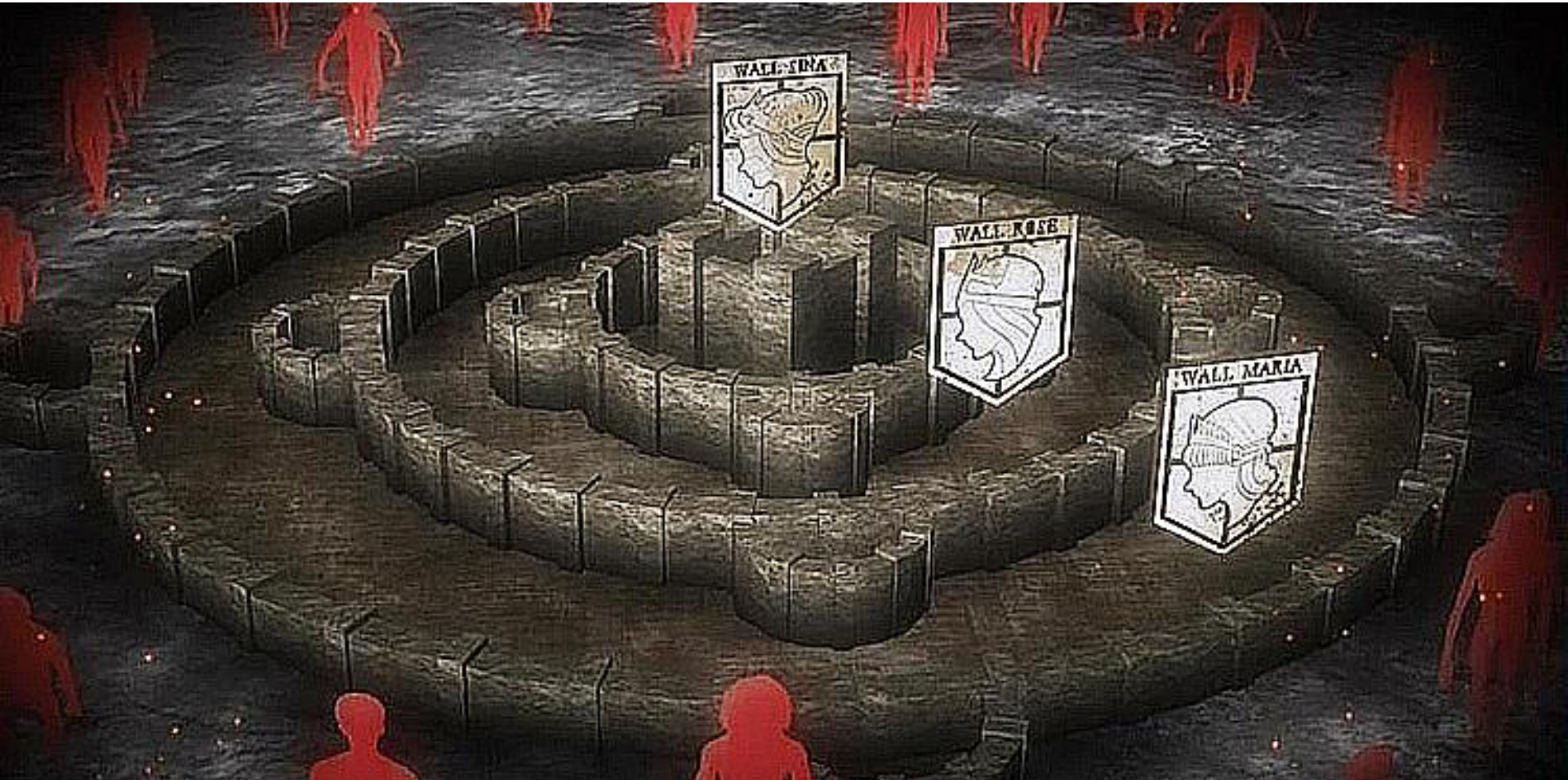
[source: Building secure software, J. Viega & G. McGraw, 2002]

The approaches to computer security

- ❑ Security by correctness 
- ❑ Security by isolation 
- ❑ Security by obscurity 
- ❑ Security by randomization 

Defence-in-depth

- ❑ Why choosing?



Mitigations

- ❑ According to the SDL
 - ❑ Designed to stop the attacker
 - ❑ If the countermeasure does not stop the attacker, it is a vulnerable countermeasure.



- ❑ Designed to slow the attacker



Mitigations

- Good effectiveness / effort balance

- Can be located at...



- ... the compiler

- ... the operating system

- ... the hardware



Compiler mitigations for buffer overflows



LLVM / CLANG



Safer function calls



- Enable warnings:
 - Warray-bounds**: Compile time out of bounds checks
 - Wformat=2 -Wformat-security**: Format string warnings

Safer function calls



- ❑ -FORTIFY_SOURCE (buffer overflow detection).
 - ❑ works by computing the number of bytes that are going to be copied
 - ❑ provides buffer overflow checks for the following functions (and wide character variants):
 - ❑ *memcpy, mempcpy, memmove, memset, strcpy, stpcpy, strncpy, strcat, strncat, sprintf, vsprintf, snprintf, vsnprintf, gets.*
 - ❑ argument consistency is checked

Safer function calls



- ❑ **-D_FORTIFY_SOURCE=1** → checks that shouldn't change the behavior of conforming programs are performed. Checks at compile-time only.
- ❑ **-D_FORTIFY_SOURCE=2** → some more checking is added, but some conforming programs might fail. Checks at compile-time and runtime.



Safer function calls

- ❑ /WE4789
 - ❑ Warns about buffer overrun when specific C run-time (CRT) functions are used, parameters are passed, and assignments are performed, such that the data sizes are known at compile time. This warning is for situations that might elude typical data-size mismatch detection.
 - ❑ *strcpy, memset, memcpy, wmemcpy*



Safer function calls

- ❑ Replace with secure version of the functions

```
#define _CRT_SECURE_CPP_OVERLOAD_STANDARD_NAMES 1  
#define _CRT_SECURE_CPP_OVERLOAD_STANDARD_NAMES_COUNT 1
```


Stack Canaries

- ❑ The compiler place a value before the return address when a function is called and check that the value has not changed when the function finalize
- ❑ Terminator Canaries
- ❑ Random Canaries
- ❑ Random XOR Canaries



Stack Canaries



- ❑ Deprecated:
 - ❑ StackGuard
 - ❑ ProPolice (reorders variables)

- ❑ **-fstack-protector**: Buffer size > 8B && functions that call alloc
- ❑ **-fstack-protector-explicit**: stack_protect attrb.
- ❑ **-fstack-protector-all**: All 😊 🕒
- ❑ **-fstack-protector-strong**: paranoid conditions (good performance)

Stack Protector	Stack Protector Strong
Function calls <code>alloca()</code>	If any of its local variable's address is taken, as part of the RHS of an assignment
Function contains buffers larger than 8 bytes	If any of its local variable's address is taken as part of a function argument
	Or if it has an array, regardless of array type or length
	Or if it has a struct / union which contains an array, regardless of array type or length
	Or if function has registered local variables



Stack Canaries

- ❑ AddressSanitizer (en gcc desde 4.8)
 - ❑ **-fsanitize=address**

- ❑ UndefinedBehaviorSanitizer (UBSan)
 - ❑ **-fsanitize=undefined** All kind of undefined behaviours
 - ❑ **-fsanitize=integer** undefined or suspicious integer behavior
 - ❑ **-fsanitize=nullability** While violating nullability does not have undefined behavior, it is often unintentional
 - ❑ **-fsanitize=bounds** Detects out-of-bounds access of arrays.
 - ❑ **-fsanitize=bounds-strict** Enables strict checking

- ❑ ThreadSanitizer
 - ❑ **-fsanitize=thread** Detects data races





Stack Canaries

- ❑ Microsoft Visual C++ 2003
 - ❑ **/GS** Stack-Based Buffer Overrun Detection

- ❑ Microsoft Visual C++ 2005
 - ❑ Buffers reordering
 - ❑ Parameter Shadowing

- ❑ Microsoft Visual C++ 2005 SP1
 - ❑ **#pragma strict_gs_check(on)**
 - ❑ More aggressive heuristics

Without GS in Visual Studio 2003



With GS in Visual Studio 2008





Stack Canaries

- ❑ Microsoft Visual C++ 2010
 - ❑ wider scope of protected functions
 - ❑ optimize away the unneeded security cookies
 - ❑ disable for specific functions with `__declspec(safebuffers)`
 - ❑ choose different level of GS protections through `/GS:n`:
 - ❑ `/GS:1` VC++ 2005 and 2008
 - ❑ `/GS:2` widened scope (default)

- ❑ Microsoft Visual C++ 2011
 - ❑ Detects range violation

```
buf[cch] = '\0';
```



```
if(((unsigned int) cch) >= MAX) {
    __report_rangecheckfailure();
}
buf[cch] = '\0';
```

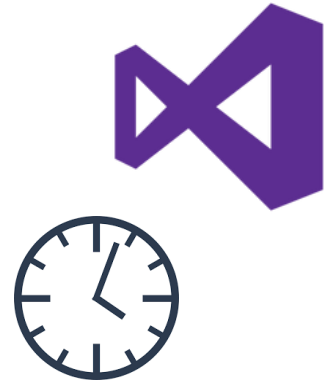


Stack Canaries

- ❑ **/SAFESEH**
 - ❑ /GS does not protect exception handler records
 - ❑ Instead of protection the stack (by putting a cookie before the return address), modules compiled with this flag will include a list of all known addresses that can be used as exception handler functions.
 - ❑ If an exception occurs, the application will check if the address in the SEH chain records belongs to the list with "known" functions, if the address belongs to a module that was compiled with SafeSEH. If that is not the case, the application will be terminated without jumping to the corrupted handler.

Stack Canaries (not exactly)

- ❑ **/RTC** Runtime error checks
 - ❑ /RTCs: stack-frame runtime error checking
 - ❑ /RTCu: variable used before initialization
 - ❑ /RTCc: value assigned to a smaller data type
 - ❑ /RTC1 === /RTCsu



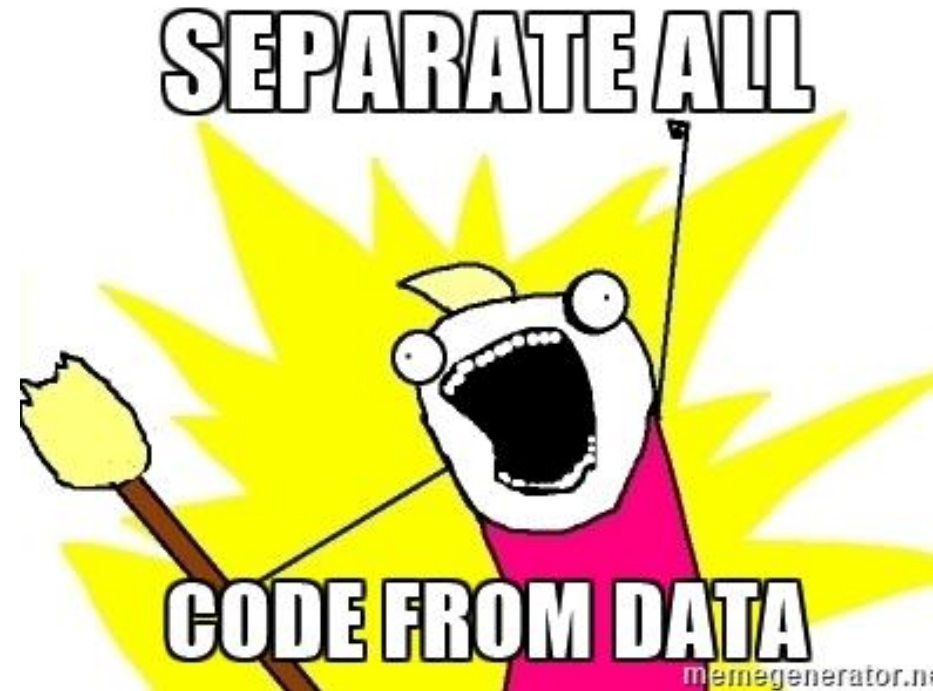
Bypassing Stack Canaries

- ❑ Incorrect implementations
- ❑ It can be a statistical problem
- ❑ Windows: SEH overwriting
 - ❑ Protected by SafeSEH and so on...



Non-executable stack (DEP)

- ❑ Code is code and data is data
- ❑ Hardware mechanism widely deployed (every computer since 2001)



Non-executable stack (DEP)

- ❑ Enabled by default in all modern compilers



`-z,noexecstack, -z,noexecheap`



`/NXCompat`

Bypassing Non-executable stack (DEP)

- ❑ Save the payload in the heap
- ❑ Return into libc attacks
- ❑ ROP

RETURN ORiented PROGRAMMING
IS LIKE a ransom note, BUT
instead Of cutting out LETTERS
from magazines YOU ARE
cutting out INSTRUCTIONS FROM
TEXT SEGMENTS

ASLR

- ❑ Address Space Layout Randomization
 - ❑ The code is loaded in different memory regions each time
 - ❑ Implemented by the operating system
 - ❑ To be of any use, you must also have DEP enabled

- ❑ But code needs to be “position independant”

ASLR



- ❑ Compiled with
 - ❑ **-fPIE -pie** for binaries
 - ❑ **-fPIC** for shared libraries.

ASLR

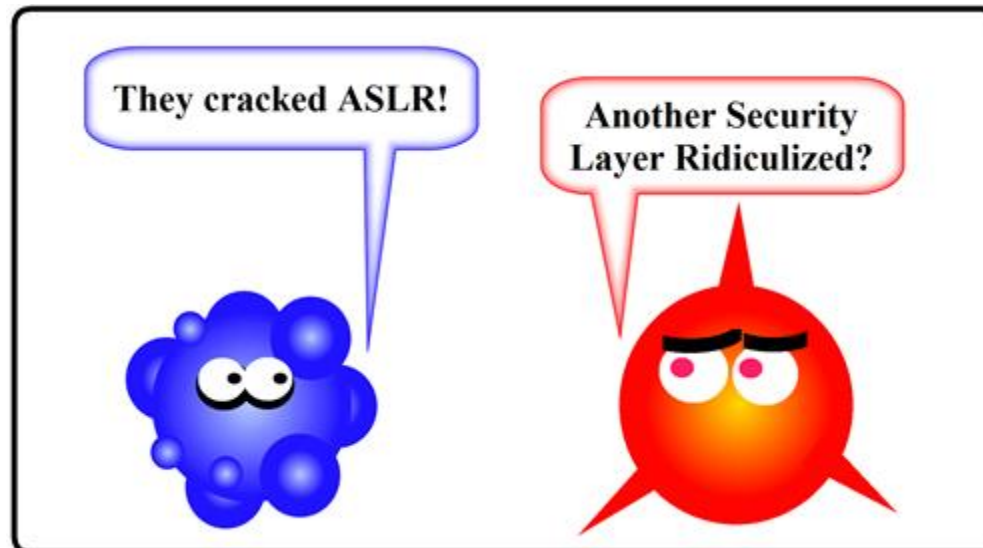


- ❑ By default, Windows® will only juggle system components around. If you want your image to be moved around by the operating system (highly recommended), then you should link with:
 - ❑ **/DYNAMICBASE** (since VS 2005 SP1)
 - ❑ **/HIGHENTROPYVA** (since VS 2012) uses ASLR with 64 bits addresses

- ❑ It also randomizes the stack

Bypassing ASLR

- ❑ It could be an statistical question (again)
- ❑ Maybe not all the libraries are randomly loaded



Control Flow Integrity

- ❑ **Restricts the control-flow** of an application to valid execution traces. CFI enforces this property by **monitoring the program at runtime** and comparing its state to a set of precomputed valid states. If an invalid state is detected, an alert is raised, usually terminating the application.
- ❑ CFI **detects control-flow hijacking** attacks by limiting the targets of control-flow transfers. In a control-flow hijack attack an attacker redirects the control-flow of the application to locations that would not be reached in a benign execution, e.g., to injected code or to code that is reused in an alternate context.



Control Flow Integrity

- ❑ **-fsanitize=cfi**
- ❑ Optimized for performance
- ❑ To allow the checks to be implemented efficiently, the program must be structured such that certain object files are compiled with CFI enabled, and are statically linked into the program. This may preclude the use of shared libraries in some cases.
- ❑ **-fvisibility=hidden** otherwise would disable CFI checks for classes without visibility attributes



Control Flow Integrity

- ❑ Control Flow Guard: operates by creating a per-process bitmap, where a set bit indicates that the address is a valid destination. Before performing each indirect function call, the application checks if the destination address is in the bitmap. If the destination address is not in the bitmap, the program terminates.
- ❑ `/guard:cf` linker flag (VS2015)
- ❑ Requires OS support: Windows 10 or 8.1 U3

Virtual Table Verification

- ❑ C++ polymorphism → vtables
- ❑ An attacker could exploit an use-after-free error to hijack the vtable using heap spraying (80% attacks)
- ❑ Detects modifications in the vtable



Virtual Table Verification

Celement::`vtable`

VirtualMethod1

VirtualMethod2

...

vtguard

- ❑ Extra entry added to vtable
- ❑ ASLR makes this entry's value unknown to the attacker
- ❑ Check added:
 - ❑ if `vtable[vtguard_vte] != vtguard` then terminate the process

Virtual Table Verification



- ❑ gcc > 4.9
 - ❑ **-fvtable-verify=std**
 - ❑ **-fvtable-verify=preinit**

- ❑ Much more complex implementation by Google team
 - ❑ Not dependent on ASLR

- ❑ <https://gcc.gnu.org/wiki/cauldron2012?action=AttachFile&do=get&target=cmtice.pdf>

Other compiler options



- ❑ Partial RELRO **-Wl,-z,relro:**
 - ❑ non-PLT GOT is read-only
 - ❑ the ELF sections are reordered so that the ELF internal data sections (.got, .dtors, etc.) precede the program's data sections (.data and .bss)
- ❑ Full RELRO **-z,now:** tell the dynamic linker to resolve all symbols when the program is started, or when the shared library is linked to using dlopen. Improves the effectiveness of RELRO
 - ❑ the entire GOT is also (re)mapped as read-only
- ❑ **-ftrapv:** Generates traps for signed overflow (may interfere with UBSAN)

Other compiler options



- ❑ **-mmitigate-rop**: Attempt to compile code without unintended return addresses, making ROP just a little harder.
- ❑ **-z,nodlopen** and **-z,nodump**: Might help in reducing an attacker's ability to load and manipulate a shared object.
- ❑ **-fomit-frame-pointer**: difficult reversing and debugging?
- ❑ **-fstack-check**: Prevents the stack-pointer from moving into another memory region without accessing the stack guard-page.
- ❑ **-Wall -Wextra**: enables many warnings

Other compiler options



- ❑ **--analyze:** performs various analysis of LLVM assembly code or bytecode and prints the results on standard output



Other compiler options

- ❑ **/INTEGRITYCHECK** places a flag in the binary that instructs the loader to verify the module's signature at load time.
- ❑ **/HOTPATCH** Enables binary hot patching
- ❑ **/SDL** enables a superset of the baseline security checks
 - ❑ enables some warnings as errors:
 - ❑ enables the strict mode of /GS run-time buffer overrun detection,
 - ❑ performs runtime limited pointer sanitization
 - ❑ automatically initializes all class members to zero on object instantiation
- ❑ **/ANALYZE** Enterprise static code analysis (freely available with Windows SDK for Windows Server 2008 and .NET Framework 3.5).

See <https://randomascii.wordpress.com/2011/10/15/try-analyze-for-free/>

Conclusions

- ❑ Compilers provide built-in defenses to mitigate overflows
- ❑ Techniques are more powerful when used together
- ❑ Attackers always develop techniques to bypass the new countermeasures
- ❑ Stay updated!



Backup slides

- ❑ Beware of optimization unstable code!

```
void getPassword(void) {
    char pwd[64];
    if (GetPassword(pwd, sizeof(pwd))) {
        /* Checking of password, secure operations, etc. */
    }
    memset(pwd, 0, sizeof(pwd));
}
```

Backup slides

- ❑ Beware of optimization unstable code!

```
char *buf = ...;
char *buf_end = ...;
unsigned int len = ...;
if (buf + len >= buf_end)
    return; /* len too large */
if (buf + len < buf)
    return; /* overflow, buf+len wrapped around */
/* write to buf[0..len-1] */
```

Figure 1: A pointer overflow check found in several code bases. The code becomes vulnerable as gcc optimizes away the second if statement [13].

Contact data

jtsec: Beyond IT Security

Avenida de la Constitución 20, Of. 208

CP 18012 Granada – Spain

hello@jtsec.es

@jtsecES

www.jtsec.es

