Co-funded by the
Erasmus+ Programme
of the European Union

ISSES

ISSES – Information Security Services
Education in Serbia

Supported by the Erasmus+ Capacity Building in the field of Higher Education (CBHE) grant
N° 586474-EPP-1-2017-1-RS-EPPKA2-CBHE-JP

MEMORY ATTACK MITIGATION

COMPUTER SECURITY

Lecture 10

ISSES 2017-2020
Novi Sad, Serbia

Information Security Services Education in Serbia (ISSES)

10.1 Memory management attack mitigation

Mitigation

- It’s now crystal clear that memory is vulnerable.
- It’s the consequence of Von Neuman’s architecture, fundamentally: all the code and all the data is in one big pile.
- It’s inevitable that the data and the code will interact in some ways and some of those are not secure.
- Discovering that all it took was a single possibility of incorrect memory usage (a virtual certainty when coding C) for any piece of software to become vulnerable came as unpleasant shock when this problem first came to light.
- Just agreeing to write better code, of course, didn’t work.
- A systemic solution was needed.

Mitigation

- The first line of defense was trying to establish a firewall of sorts around potentially compromised software.
- The principle of least privilege was put into practice and remains in use today.
- This manifested most of all by creating a separate user for any outside-facing software that has only those privileges that this software needs and nothing past that.
- This means that even in the case of a complete break where shell access is achieved not much is gained.
- This helped but any access is still a wedge that can be used to lever one’s way into the system.
- What would be ideal if the system defended itself somehow.

Prongs of attack

- **Data as code.** The reason most of these attacks work is that we can send something as user input to be stored as data, and then ask that it be executed later. Can we somehow permanently mark something as data and *not* code?
- **Internals access.** The memory layout of a program is predictable. The compiler puts things in predictable places and there's no reason for it not to: linear address space is designed with this in mind. However, this gives attackers details on the functioning of our code that they should not have. Can this be changed?

Prongs of attack

- **Tripwires.** Buffer-based attacks by necessity clobber large areas of memory. That's how they work, almost by definition. This means that they probably break things that weren't intended to be touched. Can we use this to detect faults and stop the attacked program before it does any damage?
- **Common weaknesses.** Memory faults follow common patterns. Frequently it is a matter of using functions that have no security to them, like strcpy, for instance, or gets. *Anything* that writes to some location without an explicit write limit is insecure. That's detectable. Is it possible to either persuade coders not to do these things or even automatically fix their code?

Prongs of attack

- Needless to say all these questions can be answered with ‘yes’ and all of them have been implemented to a near-universal degree.
- In a sense, it’s one of the largest security responses that have been made in recent times: this class of vulnerabilities has been such a problem that alterations to hardware architecture, alterations to operating systems, and alterations to compilers have been made to combat it.
- We’ll discuss examples of all of these in turn, focusing on an Intel CPU running Linux and using a GCC compiler.
- Equivalent alterations are present in other circumstances as well.

Relationship with other lectures

- Lectures 8 and 9 are of particular importance as those define what it is, precisely, we'll be defending against.
- Lecture 8 is particularly crucial as it explains what it is we modify to get this new, enhanced security.
- Lectures 2 and 6 are also important as what's presented here can be seen as an extension of physical/hardware security and kernel adaptations.
- Remind yourself of the concept of security feature management from lecture 7 is also important as some of these features can be toggled on and off.

Information Security Services Education in Serbia (ISSES)

10.2 NX/XN BITS AND PAGING ALTERATIONS

Paging access rights

- When dealing with paging each access to a linear address is either in supervisor mode or in user mode.
- This is a collapse of privilege levels used before: instead of rings 0-3, we have, in essence 0 and 3.
- This a pattern you will see in CPU feature development: what doesn't get used gets removed.
- The level of access comes from the CPL field in the CR4 register, as we've discussed before. For CPL < 3, the access is supervisor-mode, for CPL = 3, the access is user mode.
- Addresses, likewise, are either user-mode or supervisor mode based on the value of bit 2 in *any* of the paging structures leading up to it.

Paging structure bits

N	Name	Purpose
0	P	Present: whether it is in memory.
1	R/W	If 0, allows only reading, not writing.
2	U/S	If 0, no user-mode accesses.
3	PWT	Controls caching
4	PCD	Controls caching
5	A	Accessed: used as bookkeeping information for long-term scheduling.
6	D	Has ben written to: used as bookkeeping for long-term scheduling
7	PAT	Controls caching
8	G	Global translation control: allows everyone to see a page irrespective of PCID and its TLB.
62:59	Key	Protection key
63	XD	eXecute Disable; If 1, instruction fetches are not allowed from this page.

XD bit

- The XD bit is a very simple innovation, but it provides a great deal of security.
- If it is set to 1, it means that any instruction fetch from this location will fail with a page fault.
- This limits the use of a page so marked as only data, not code.
- This is an immediate and excellent protection against arbitrary code execution as anything where we store potentially dangerous data can be made such that it *can't* be executed.
- This means that the standard stack attack is no longer possible without finding a way around XD.

XD Permission matrix

Value of XD bit at position 63				Can use as:
PML4	PDP	PDE	PTE	
1	Not checked	Not checked	Not checked	Data
Not checked	1	Not checked	Not checked	Data
Not checked	Not checked	1	Not checked	Data
Not checked	Not checked	Not checked	1	Data
0	0	0	0	Data/Code

As can be seen, the value of the XD bit propagates: if it is set at an earlier place it will propagate down to lower levels.

Quick terminological note

- It should be noted that ‘XD’ is what Intel calls this technology.
- The generic term is ‘NX’, and it is the one used by AMD.
- ARM calls it XN ‘execute never.’
- Other platforms have similar solutions: the central idea is that there’s a way to mark an area of memory as never containing code.
- In a sense this is an explicit denial of the Von Neuman architecture at its core, installing a virtual separation into code memory and data memory.
- This means you cannot make self-modifying code.
- This is probably a good thing.

Silver bullet?

- Does XD always work? Is it the end of arbitrary code execution?
- In a word: no.
- As an example: CVE-2014-5439
- The sniffit network packet sniffer in version 0.3.7 can drop the user in a root shell with a malformed configuration file.
- Why root? Because sniffit installs are nearly always setuid'd because you need root access for that level of network access.
- And how? Buffer overflow vulnerability (a bad one) and sequence of ROPs.

Find the vulnerability

```
char *clean_string (char *string) {
    char help[20];
    int i, j;
    j=0;
    for (i=0;i<strlen(string);i++) {
       if( (isalnum(string[i]))||(string[i]=='.') ) {
          help[j]=string[i];
          help[j+1]=0;
       }
       j++;
    }
    strcpy(string, help);
    return string;
 }
```

XD/NX support

- For this to work, support from both the OS and the compiler is needed.
- After all there is no way for the CPU to know if given memory is going to be code or not without being told.
- This is hinted by the compiler and then conveyed through the operating system.
- That means that how code is compiled directly impacts how well the protection works.
- Miscompiled code is easily vulnerable especially in fraught situations when 32-bit libraries are called from 64-bit code and, for instance.

Miscompiled code

- CVE-2014-0892
- IBM Notes and Domino versions on 32-bit platforms use incorrect gcc options and are vulnerable to attacks that bypass XD protection.
- What's the problem? Someone got confused and submitted the –z execstack option to gcc.
- This gives the code the permanent property of being able to read something giving you also access to execute it.
- In other words: it straight-up disabled NX.
- As a result, this code *and all processes that are run from it* won't have NX.

Checking for NX support

- If you have a recent version of Linux, you have NX support built in.
- And if you compile your own code, you can make sure you use the correct options.
- Luckily, gcc turns it on by default.
- However, if you have code someone else compiled (a situation that just about everyone is in) you have to have some method to check if the code is valid or not.
- How? Luckily all this information can be obtained using the readelf tool.

Checking for NX support

```
~/prg/sec/stack ▷ gcc -z execstack -o vulnerable vulnerable.c
~/prg/sec/stack ▷ readelf -W -l vulnerable|grep 'GNU_STACK'|grep 'RWE'
  GNU_STACK      0x000000 0x0000000000000000 0x0000000000000000 0x000000 0x000000 RWE 0x10
~/prg/sec/stack ▷ readelf -W -l /bin/ls|grep 'GNU_STACK'|grep 'RWE'
~/prg/sec/stack ▷
```

It's evident what we are doing: we are using the –W option to be able to read wide lines efficiently. –l shows segment headers which is where we'll find information regarding the parts of the executable which naturally includes 'GNU_STACK' which is our stack.
Then we grep for the existence of a 'RWE' string which signifies that the stack has 'Read', 'Write' and 'Execute' permissions which means it is vulnerable should a buffer overload be encountered.

Information Security Services Education in Serbia (ISSES)

10.3 Address space randomization

Reliable internal state

- The only way we can launch dedicated attacks based on memory layout is if we *know* the memory layout.
- The memory layout of a process is mostly arbitrary and there is no pressing reason it should be predictable.
- However, it’s easier for everyone if it maps reliably to the same addresses and this was the default behavior for most of computing’s history.
- This predictability means that an attacker has a fair chance of guessing where which part of memory is, much in the way we have done when we were implementing our stack attack.

Randomization

- A simple solution presents itself: If we make sure we *can't* predict the memory layout of a process, that clearly sabotages any attempts at memory manipulation.
- This isn't that difficult. The process must define its sections, and into which segments it purports to load those segments as a part of any basic executable format.
- This is what we read when we use the readelf command.
- All that's required is, when loading, randomizing those.
- Support is required, and certain legacy software may make unwarranted suppositions about the layout, but it's simple enough to implement.
- A feature much like this is part of every modern general-purpose operating system.

GDB frame info without address space randomization

```
Breakpoint 1, 0x0000555555555149 in main ()
(gdb) info frame
Stack level 0, frame at 0x7fffffffe400:
 rip = 0x555555555149 in main; saved rip = 0x7ffff7dfccb2
 Arglist at 0x7fffffffe3f0, args:
 Locals at 0x7fffffffe3f0, Previous frame's sp is 0x7fffffffe400
 Saved registers:
  rip at 0x7fffffffe3f8
(gdb)
```

First run

```
Stack level 0, frame at 0x7fffffffe400:
 rip = 0x555555555149 in main; saved rip = 0x7ffff7dfccb2
 Arglist at 0x7fffffffe3f0, args:
 Locals at 0x7fffffffe3f0, Previous frame's sp is 0x7fffffffe400
 Saved registers:
  rip at 0x7fffffffe3f8
```

Second run

GDB frame info with address space randomization

```
Stack level 0, frame at 0x7ffdc704ef20:
 rip = 0x555885b7c149 in main; saved rip = 0x7f756537fcb2
 Arglist at 0x7ffdc704ef10, args:
 Locals at 0x7ffdc704ef10, Previous frame's sp is 0x7ffdc704ef20
 Saved registers:
  rip at 0x7ffdc704ef18
(gdb)
```

First run.

```
Stack level 0, frame at 0x7ffcd1dedad0:
 rip = 0x56458211a149 in main; saved rip = 0x7f85e82c7cb2
 Arglist at 0x7ffcd1dedac0, args:
 Locals at 0x7ffcd1dedac0, Previous frame's sp is 0x7ffcd1dedad0
 Saved registers:
  rip at 0x7ffcd1dedac8
```

Second run

ASLR implementation

```
unsigned long arch_align_stack(unsigned long sp)
{
        if (!(current->personality & ADDR_NO_RANDOMIZE) && randomize_va_space)
                sp -= get_random_int() % 8192;
        return sp & ~0xf;
}
```

- This is from the x86 version of process.c, the implementation on how to make a process in Linux.
- You will find it in /arch/x86/kernel/process.c, line 901 in Linux v5.10.9 kernel source.
- This of course not the entire implementation (various other adaptations are scattered across the codebase) but it does show how the position of the stack is randomly shifted at the making of a process in the code which normally aligns the stack.

Detecting ASLR

- You can check if ASLR is on in Linux by querying the procfs file system.
- On the path of /proc/sys/kernel/randomize_va_space is the controlling parameter.
- Read, it provides the current value of the parameter, and it can be overwritten to change the system-wide ASLR setting, though it should be noted that this change persists only until next reboot.
- Valid values for it are:
  - **0** – ASLR disabled.
  - **1** – ASLR enabled, but not for dynamically allocated memory.
  - **2** –ASLR enabled, even for dynamically allocated memory.
- The modern default is ASLR=2

ASLR for dynamically allocated memory

- This is done so that every time the program adds to its data segment by allocating more memory through the movement of the *program break* (all achieved through the brk system call) it will randomly move the allocated memory.
- It does this using the randomize_page function which aligns a given address, and then places it at an equally aligned point within a given range of permissible addresses.
- In case of x86 randomization this is any page-aligned value within 0x02000000 of the original address. This level of freedom is acceptable because the linear address space is vast and there's plenty of space to hide our data in.

Managing ASLR

- Normally, on a given system the thing we want to do is engage ASLR if it isn't on by default, as it likely is.
- This can be safely done through **/etc/sysctl.conf** where **kernel.randomize_va_space** should be assigned the desired value.
- Temporary systemwide changes can also be achieved through the **/proc/sys/kernel/randomize_va_space** special file.
- If running something through GDB, there's an in-built command to control behavior through **set disable-randomization on/off**.

Managing ASLR

- However, for experimentation, the handiest function is ‘setarch.’
- The primary purpose of setarch is to change the architecture the targeted program sees as running under.
- Therefore, if you were to run setarch i686 <programname> you would get programname to see itself as running under the i686 architecture type as opposed to whatever the machine actually is (these days, typically x86_64 on PC).
- However, it can also assign to processes it runs *personalities.*
- Personalities are, in essence, system architecture features and modes of operation which may be enabled or disabled.

Personalities

- While there is a great number of personalities available (you can find out more about them with man 2 personality on any Linux machine: googling the command will inevitably bring up much the same) only a few are of particular interest to us.
- These are ADDR_COMPAT_LAYOUT (achievable through the –L option of setarch), and in particular ADDR_NO_RANDOMIZE (achievable through the –R option of setarch), and READ_IMPLIES_EXEC (achievable through the –X option of setarch).
- The last two are of particular interest to us, the first will, naturally, stop address randomization.

READ_IMPLIES_EXEC

- When mapping an area of virtual memory, either to a file or for other uses (such as loading shared libraries, for instance) in a POSIX environment you need to use the mmap system call.
- This system call requires that we specify what we wish to do with that memory through the desired protection level.
- This personality means that reading a page means that we must also have the right to execute it. Thus pages are either read/execute or read/write/execute.
- This used to be a default, and this is what XD/NX bits saved us from.
- We can recreate this bad behavior in order to test software in a hostile environment.

Information Security Services Education in Serbia (ISSES)

10.4 COMPILER DEFENSE

The role of the compiler

- Defense against memory attacks which rely on the destructive interplay of architecture, OS, and user code must rely on all three components to do their part in defense.
- Failure at any level can re-introduce vulnerabilities.
- Particularly dangerous is a misconfigured OS—something we've covered here—and an incorrectly used compiler.
- A compiler can produce all sorts of code: you can target any desired architecture and ask for a great deal of the compiler: engaging certain features, disengaging others and so on.

The role of the compiler

- Managing all these switches and options and similar is that it's often the job of *multiple* intervening languages and subsystems.
- To compile a fairly average modern C/C++ library or application it is typical to use CMake to process a configuration file which, in turn, will create another configuration file which make can then call and invoke the compiler with adequate options.
- This means that there's a great deal of disconnect between the programmer and what options are used, precisely, to compile a given piece of code.
- An even greater distance is between the *end user* and the options used.

Compilers and security

- CVE-2014-0892 has taught us that a single wrong option can cause old vulnerabilities to ‘come back from the dead’
- The cause of such misconfigurations can be many:
  - Legacy options nobody bothered to remove which end up overriding otherwise sensible defaults.
  - Debugging options sneaking into production code (this is more common than you think: it’s not at all uncommon to ship debug-compiled code if the programmers never bothered to compile a release configuration until the very end; discovering that it doesn’t work and having a debug version which is perfectly operational they ship that which works)
  - Unexamined fixes: sometimes a compiler option fixes a persistent bug at the cost of lessened security. Perhaps the bug was, in fact, the security mechanism trying to save the programmers from themselves.

Unexamined fixes

- It’s very common for programmers to, faced with the overwhelming complexity of configuring a modern compiler and a persistent bug to just paste whatever they found on StackOverflow into the makefile and, if it works, stop messing with it.
- *This is incredibly dangerous.*
- In order to help you be the defense against this particular type of problem, we will cover the main security options available in a modern compiler.
- We will use GCC due to its ubiquity, support on multiple platforms, and FOSS nature.
- Other compilers have roughly equivalent options you can easily find, now that you know what to look for, in their manuals.

References

- This section is primarily based on the GCC documentation itself. https://gcc.gnu.org/onlinedocs/
- You can get useful information by looking at rules for hardening imposed on various distributions.
- Rules for hardening a package for package contributors, in particular, are a good repository of security best practices or, at the very least, security *typical practices.*
- Particularly in-depth is the Debian guide accessible here: https://wiki.debian.org/Hardening

Helpful options

- Before engaging various hardening solutions and other forms of protection it's useful to make sure our own code doesn't have serious errors.
- To that end, it can be good practice to enable a wider array of warnings and other helpful compiler messages.
- Useful to consider are
  - -Wall and –Wextra
    - This will include additional warnings
  - -Wconversion
    - This will warn you for any instance of implicit conversion in your code which may affect data integrity.
  - -Wsign-conversion
    - Warn on specifically conversions between signed and unsigned that are not explicit.

Helpful options

- -Wformat-overflow
  - This warns about calls to formatted I/O functions which may cause a buffer overflow. Note the use of ‘may.’ Since it’s impossible to know how many bytes will be actually written into the buffer, a heuristic guess must be taken.
- -Wformat-security
  - Automatically detects cases where printf attacks are possible, i.e. where a non-string-literal is used in a printf without format arguments. Strictly speaking this is also fixable using string sanitization.
- -Werror
  - Turns all warnings into errors. May be going too far, since occasionally a warning is harmless and it’s rare to see a large project that compiles without warnings. Still, for security-crucial code, it may be worth it.

What warning options do?

- No amount of automated analysis will ever make code safe.
- Certainly, you should use at least some of these in your code as they may help catch something you've missed in manual review.
- But with your own code more thorough forms of code analysis are even better.
- Where these options are useful are in situations where you must rely on other people's code.
- If you must use a binary library it may be possible to recover how it was compiled. Finding warning options enabled can serve to increase your faith in the quality of the code.

General options

- -arch x86_64
  - If at all possible, compile all code for fully 64-bit execution. This is likely not to be possible all the time because of compatibility issues, but 64-bit mode beats out IA-32e (if running on a 64-bit ready CPU) on a number of security features. It also increases the size of the address space available which increases the effectiveness of ASLR.
- -pie –fPIE
  - Demands dynamical linking and position independent executable creation (the two options are one for the linker and one for the compiler) of the compiler/linker. This is improves the effectiveness of ASLR.

Security options

- These options specifically engage various mechanisms which, with the cooperation of the architecture, compiler, runtime, and OS, all working together, increase the security.
- These are generally not all on because there is a cost to their activation.
- This cost can be in compatibility (esp. compatibility with legacy code), in performance, executable size, or desired behavior.
- Used in conjunction, these options can do wonders to harden your executable against most forms of attack.
- Note that most of them can only work with system support.

Security options

- -mmitigate-rop
  - This is a now-obsolete technique to prevent ROP attacks.
  - It turns out that there's so many ways to effect a ROP style of attack that something with broader hardware support is needed.
  - x86_64 has CET and ARM has PAC.
  - CET keeps a record of all valid jump addresses on a specially protected shadow-stack and PAC cryptographically signs valid jump addresses.
  - Because of the advent of these better protection mechnanisms, GCC versions post 9.1 don't have this option, and instead enables the appropriate functionality if supported by the system, the kernel and the linked modules, especially glibc.

Security options

- -Wl,-z,ibt,–z,shstk
  - Passes specific instructions to the linker to link a given library with CET support. If you have CET enabled in your code it is an error to dynamically link to a non-CET library.
  - Note the commas: this is the syntax for passing large linker options to the linker from the interface of the compiler in GCC.
  - The parameters simply engage IBT (indirect branch tracking) and the Shadow STacK functionality.
  - You may be unable to fully utilize CET if you rely on a lot of existing code which is compiled without CET support.
  - This is a demonstration of how dependency creep can be a negative factor in security.
  - Normally you don’t have to link this directly.

Security options

- -fcf-protection=full
  - Augments the code with necessary instrumentation to protect control flow from ROP/JOP attacks.
  - This is done through CET.
  - This is the standard way to enable CET in newer versions of GCC.
  - The full value with specify that both branch and return addresses are checked for validity.
  - You can detect CET from within code through the __CET__ macro.

Security options

- -mindirect-branch=thunk
  -mfunction-return=thunk
  - This enables retpoline-type Spectre mitigation techniques.
  - Note that this is exclusive to x86 architectures. If discussing ARM it will not work.
- -fstack-protector-all
  - An option which generates additional code around each function serving to protect against stack-smashing attacks.
  - It does so by expanding the stack with a single additional guardian variable. These are initialized before execution enters the function and are checked on its exit. If the guard check falls that means someone smashed the stack.
  - Normally this only applies to some functions. –all means it applies to all, despite the performance cost.

Security options

- --fstack-clash-protection
  - This is an option needed to protect against a subtype of stack-smashing known as a ‘stack clash’ attack.
  - This attack uses the fact that the stack and heap may be (given the directions of growth) ‘clash’ by having their areas overlap.
  - It’s generally quite a serious threat: see also CVE-2017-1000367, CVE-2017-1000366, as well as CVE-2017-1000370 and CVE-2017-1000365.
  - This option may not be available on all platforms.

Security options

- -D_FORTIFY_SOURCE=2
  - Technically this is not a compiler *or* linker command. This is a feature test macro. It's a mechanism for enabling or disabling certain runtime/OS-mediated functionality.
  - Strictly speaking all this does is do a command-line macro definition (this is what –D is for) of the _FORTIFY_SOURCE macro to level 2.
  - Level one does common checks for errors that may lead to buffer overflows. It uses compile-time checks mostly.
  - Level two adds more significant run-time checks and disallows more risky behavior. Note that this means that the compiler changes code behind your back: if you use this make sure you have good tests to make sure this hasn't introduced any unexpected alterations.

Security options

- -Wl,-z,relro,-z,now
  - Instructs the linker to use read-only reallocation (RELRO) at the maximum lelvel. This is used to mark various sections of the binary (in ELF format) as read-only.
  - The purpose of this is to firewall as much of the binary structure as possible, so as to leave as little attack surface as possible.
- -Wl,-z,-noexecstack
  - Instructs the linker to mark the entire stack as non-executable. May cause compatibility issues, but protects against any attacks which slip in shellcode. It still leaves the possibility of ROP, but this helps reduce, to a significant degree, remote code execution vulnerabilities.

Security options

- -fvtable-verify=preinit
  - Only valid for C++
  - Instructs the compiler to create structures and code instrumentation which verifies at each call that pointers used to implement virtual member function calls are correctly formatted.
  - The preinit suboption is used to determine the point at which the necessary structures are built: in this instance they are built before any libraries are loaded. If it were ‘std’ the structure would be built post library initialization.

A more drastic approach

- Defending with a compiler can be taken further
- Instead of just changing how code is interpreted, it is also possible to change the language and the runtime entirely to be more secure.
- Few languages except C and C++ are suited for systems programming. Until recently no credible alternatives existed.
- Lately *Rust* has been touted as a viable alternative to C/C++.
- Performance-wise Rust is quite comparable to C/C++
- However, unless explicitly requested through the ‘unsafe’ keyword, Rust is entirely memory safe.