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 100 bugs (I think? -¿ or 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.

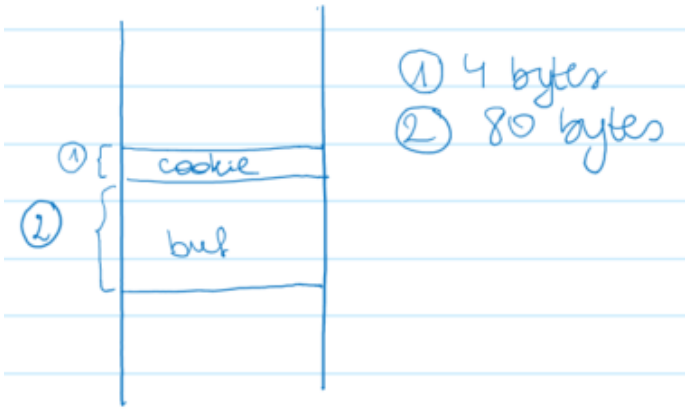

Figure 2.1: Cookie and buf allocated on the stack together.

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.

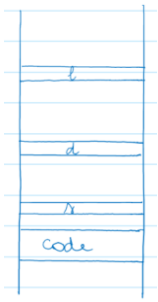

Figure 2.2: A program with s a static global variable, a local variable (l) and a dynamic variable (d).

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).

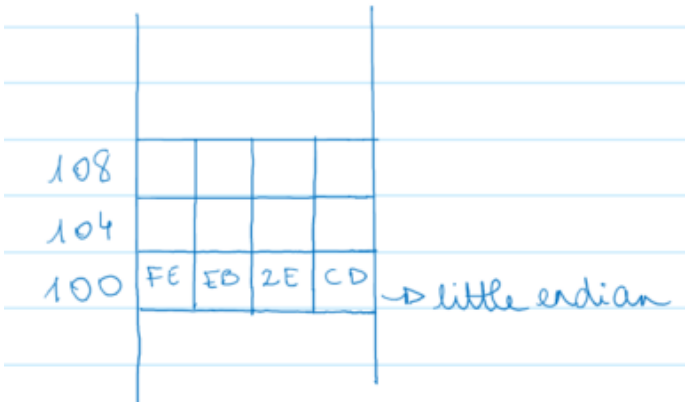

Figure 2.3: Placement of the attack code.

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,... -¿ 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.

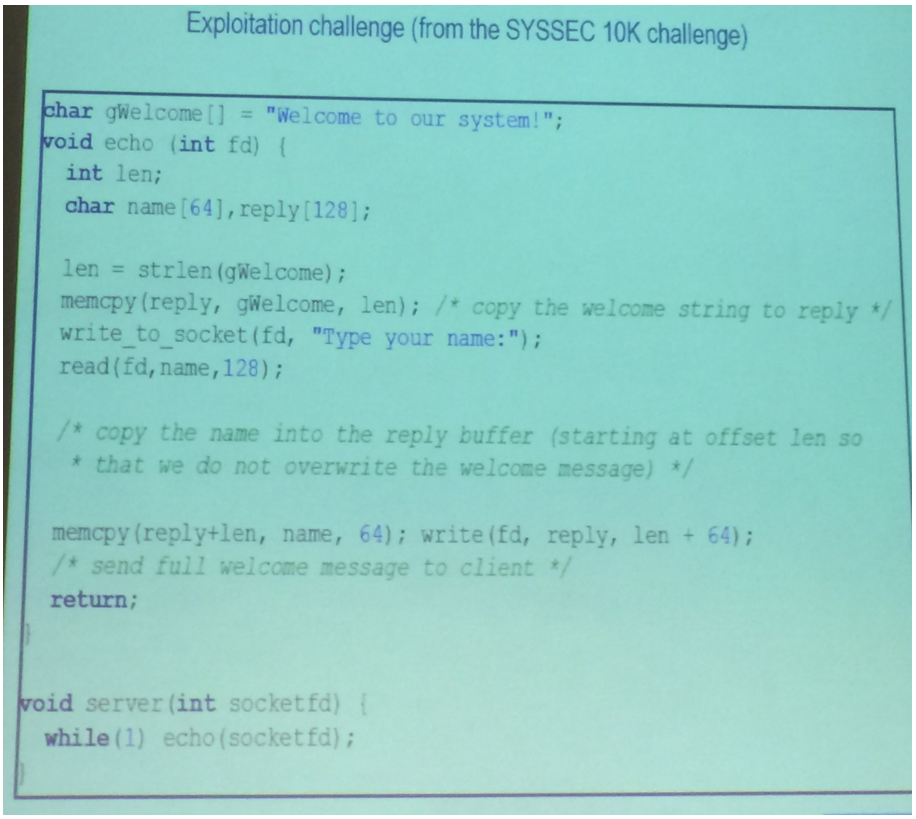

Figure 3.1: Lesson 3, Extra slide.

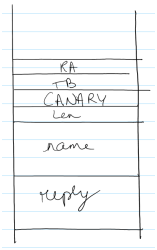

Figure 3.2: Lesson 3, Extra slide.