CS 405 Final Presentation

Mark Webster

CS 405

Professor Alhwaiti

https://www.youtube.com/watch?v=0S7EdKfy6hM

My name is Mark Webster, and this is my security policy presentation for CS 405. During this course, I created a security policy for Green Pace, a software company. It was created out of a need to have a uniform approach to secure coding and testing among the developers. This guide covers several rules for dealing with the common errors and vulnerabilities that arise during developing with C++ that can lead to larger security issues. Of course, writing secure code is just one element of security. We will strive to practice defense in depth, which will allow us to layer in security measures to mitigate any potential holes or flaws in our security system. As you see in this threat matrix, the 10 coding standards that I have identified and the problems that they listed are ranked here in order of likeliness, their priority level, unlikeliness, and their low priority level.

I'll go into depth with these a little bit later, but in short, the most likely and highest priority threats are commonly the ones that are the easiest mistakes to make with the highest remediation costs. In other words, what errors can a developer make that down the road can cause the most catastrophic or difficult to find errors? For example, if we do not sanitize input that we pass to other subsystems as in not using parameterized statements for SQL, this can cause a massive problem later on because bad input can be passed. Furthermore, we could also have unintended behavior caused by things like integer wrapping when unsigned integers are used in computation. If we do not verify first that the result does not wrap, an undesired result will be passed to our function causing again, undesired or undefined behavior. We have the standards that are solid practices, but aren't necessarily those difficult-to-fix things, like using complete logic when we write switch statements or if statements catching, trying to catch rather all exceptions in our in try-catch blocks. We don't want to leave any stone unturned. Those are things that are low remediation and easy fixes in code.

So looking at our 10 coding principles here at Green Pace, we want to validate input data. This is very similar to what I was stating about sanitizing our input data. We want to make sure that what a user is inputting aligns with what we need. So we need to have guidelines in every program for what we're looking for. And we need to make sure that the program can handle erroneous input. So it's not to cause any undesirable behavior.

Heeding compiler warnings. We need to make sure that our developers are listening to our IDEs. Whenever that compile process goes through and those mistakes are found, sometimes they're not outright mistakes. Sometimes a warning will just state, “Hey, this is something you might want to think about or watch out for”. Those are suggestions, but they shouldn't be taken as something that's just a throwaway. Those warnings are there for a reason. And the C++ IDE in our case might be pointing out something that's going to come to fruition later and cause bigger problems.

We want to architect and design for security policies. As I'll talk about later, this is a “Day One” process. We want to make sure that we are focused on everything, from our language choice to our dependencies, to how we structure our code. We want to make sure that it's implemented for maximum security.

Keep it simple, or “KISS”--keep it simple, stupid! This is a classic developer motto. There's no need for complexity if you don't have any need for that complexity: keep it simple, keep it straightforward. Added complexity and unnecessary complexity just add a greater likelihood of mistakes. Default deny: this refers to, in a security sense, how we want to deny any access to any resource by default. We want to

make sure that until a user's identity is verified and we verify that they're authorized to access a resource, they are denied from accessing it., Adhere to the principles of least privilege. This just refers to how a user should only have access to the bare minimum resources that they need to complete a certain task. An accountant does not need access to, for example, the database where all the cat photos are stored for our cat website company. We don't need people to be able to mess with things that they don't need to. This is how human exploitation happens as well as just plain old-fashioned mistakes. Someone accidentally does something with a resource. We lose it, it becomes corrupted, et cetera.

Sanitized data sent to other systems. In no manner should we be passing along any type of data that has not been looked at by our code and deemed safe. In terms of those SQL statements, we want to be using parameters. We want to parameterize those. We want to make sure that we railroad user input into the form that we need so that we can escape any malicious intent there and just have something that works in our system. Defense in depth, as I said before, that's layering in security measures so that we can mitigate any holes or flaws in our security system. No one layer is going to protect against everything, but all the layers together can form a fantastic bull work against any malicious actors using effective quality assurance techniques.

We need to have in-depth testing routines that refer to everything from a unit test on a code level to an automated penetration test that we might look at after deployment, just to make sure consistently that the security measures that we have in place are working. Something like load balance testing, stress testing. How many users can our system scale up and handle without causing problems?

We need to adopt a secure coding standard. It's the responsibility of every member of every development team to make sure that we're writing secure code. That's the reason for writing this whole guide in the first place: we need to have a universal approach that every member of every team can follow and we can all write the same secure code.

These are the ten coding standards that I focused on during writing the guide.

First, as I said before we always want to use parameterized statements when available. As you can see that one is one of the most important things in coding, especially when it comes to development where we're going to be tapping into databases; we're going to be accessing our information. I've ranked these in order and that one is at the very top because that is incredibly lethal if someone should gain access to our databases via injection. We want to, second, guarantee that strings have sufficient space for character data and the null terminator. So string literals and strings do utilize different buffers. So a string has on paper an unlimited buffer space because if it expands past the original buffer that's been allotted, the system will just allot a new buffer, remove the information over, and delete the old buffer. However, character arrays and string literals will have limits on the buffer size. And we want to make sure that we're not overflowing as that can cause memory problems. We could potentially overflow into another memory area and corrupt data there. It could cause a lot of undesired behavior. That's why buffer overflow is one of the biggest exploits that malicious actors use.

We want to ensure that unsigned integer operations do not wrap. So that can cause again, as I said, undesired behavior if an unsigned integer is pushed over its max value through some form of computation. We need to understand that it could potentially wrap. In other words, it can go from the maximum value back around to the minimum value.

We need to be aware that that can cause a lot of problems that are hard to pinpoint. Sometimes if you have a syntax error or some sort of logic error in a code, those errors are a lot easier to find than a computational error where the code is written correctly, but the outcome is not desired. We want to use pointers to const when referring to string literals. String literals are by default unmodifiable–they're immutable. So by using that constant keyword we can identify them as immutable from the get-go. So we can't modify them. If we try to change them, there'll be some type of thrown exception and we won't run into any fatal errors that will cause a program to exit.

Free dynamically allocated memory we no longer need. This simply refers to being aware of the lifespan of the objects that we're using in code. If we no longer need it, go ahead and deallocate that memory. We need to make sure that we are keeping a very streamlined memory approach. We don't want the stack to be cluttered with the remnants of objects that are just taking up space because they are no longer being used. We want to streamline.

That goes hand in hand with the next point, which is to minimize the scope of variables and functions. We want a variable and function to have essentially the minimum viable scope that it needs. If something doesn't need to be global, it shouldn't be declared as global. If something is only going to be used in a small function, it doesn't need to continue existing once that function goes beyond its life lifetime.

We also want to utilize try-catch blocks to catch and handle all exceptions that may occur. We don't want any thrown exceptions being unhandled. Again, this can cause undesired behavior. It can potentially break the program. Plus, we want to plan for all cases. We need to understand that sometimes things don't go as planned and the code needs to consider that. We need to have measures in place to handle all possible outcomes. In terms of assertions, we should be using assertions before we enter functions, while we're in functions, and after a function executes to check all those parameters and results and to make sure that everything is following the logic that we want it to. That goes back to striving for logical completeness: catching all exceptions, making sure that our if-then statements and our switches take into account everything that can happen.

We don't need to get down to a granular level on those. Sometimes we just need to have in place, “If these three things don't happen, whatever else happens, do this” so at the end of the day, we can cover all of our bases. And then finally, just a small syntactical error that sometimes people make: treating relational inequality operators as if they are non-associative. The standard relational inequality operators do not do any assignment. We want to use parentheses to make sure we take that left-to-right read schema into play so that we're not causing any weird undesired behavior, similar to what I was talking about with computational problems. Sometimes the code will look like it's written correctly, but if we have misplaced or grouped things in the wrong order–well, if the computer doesn't know the difference, it'll just go ahead and compile and read that and do what it thinks we want it to do. So make sure that we are treating those operators in their correct form and grouping them accordingly.

We want to focus at some point as well on encryption. There are three main forms of encryption that we are going to want to work on here at Green Pace. There's encryption at rest. Now this refers to how data is encrypted as we're writing it into storage. And also as we're decrypting it, as we're reading it out of storage. We want to protect the data on disk. If we have, I would argue all of our data should be

encrypted, not just the data that we deem sensitive, because any data in the hands of a malicious actor potentially can be used in an exploitative manner. We could use the key methods, we want to use encrypt solid encryption keys, possibly.

We could even partition data into separate little chunks and have those encrypted with different keys so that if someone gets access to even one to, even one key they can't just unlock everything that there's just a small partition that they can access encryption at flight. When we are transmitting data across networks, it needs to be encrypted. This is actually, the most open our data is going to be to exploitation by malicious actors. There are so many ways that those packets being transmitted across the internet can be intercepted or altered or sniffed. We just want to make sure that any data that's being moved needs to be encrypted. We can use always-on encryption as our standard: nothing should go across in plain text format. And finally, we have encryption in use. This refers to the ability to access and perform tasks with our encrypted data. Now, this used to be a little bit more complicated: there would be hardware execution environments, et cetera. Now we have more modern means where we can just operate directly on encrypted data without having to do any type of virtual decryption, et cetera. We can just do computation and data sharing on those as though they're plain text. Making sure that our encryption in use is on ensures that the data is encrypted at all stages.

Let's talk about a Triple-A policy. We want to authenticate, we want to check authorization, and we want to make sure we have accounting so authentication. We just want to verify that if I access a resource, I am who

I say, say I am I said, commonly, we use a username and password schema. We can expand that to so many different types of security. We can use two-factor authorization. I personally love 2FA. At fist, I thought it was a big pain when my various apps that would require me to enter a number that I would get via text message or use something like Google authenticator to enter a randomly generated string. But that's fantastic. We could use geolocation to verify that, “Wait a second, you're accessing from a location we don't recognize. Let's do a little bit deeper authentication”. We want to just build the strongest lock on our doors possible.

In terms of authorization, once a user gets into our system, they shouldn't just have access to all the resources. We need to have some sort of RBAC or IBAC where users are limited in what they can access. As I said before, with the principle of least privilege, they have access to the things they need to do their tasks and nothing else. We want to make sure that resources remain partitioned and the best way to do that is just default deny. If anyone tries to access anything outside their authorizations, deny. In terms of accounting, this is important because we need to have logs of what's going on in the system. We need to have the bird's eye view and the ability to go down to a finer detail level. So what users are on the system, and what are they accessing? What are they modifying, where are they located, et cetera? In terms of security, accounting acts as a paper trail in case anything does happen.

We can go back and identify patterns. We can see that they were in the system for 30 minutes. They accessed this and this information has been compromised. We can use it to trace back to how they got in; what was the weakness in our system? Did someone just write down a password or username on a post-it note and accidentally leave it out in the open or was this some sort of something more intricate?

Shifting focus, these are a series of unit tests that I wrote, just to see if we could break the logic of some code that we were working with. This is the kind of thing that we're going to want to see on a more consistent basis with our coding. Unit testing goes hand in hand with writing code. Writing code is half the battle: testing is the other half. They should be done concurrently.

With this example right now we are testing that if you try to resize a collection with a negative number, it'll throw an error. So we were using the Google assertions logic here. We've got a expect any throw call here. When a collection resizes with that negative one parameter, it should throw an error because we can only resize with a positive number. As far as I'm aware, there's no such thing as a negative-one size array. As we can see here,the test was run and the test passed, which means that our expect-any throw was fulfilled. When we tried to resize the negative number, it did throw an exception which was caught.

So we wanted to, in this one, check to see that the reserve function increased the capacity, but not the size. In other words, we wanted to increase the maximum number of elements that can be inside of our collection, but not necessarily the size of the collection. The size of the collection is actually the number of elements that are present within our collection. So I did enter some logic here at the very beginning, because I wanted to make sure that if the size was zero, we would just start with a few entries in there. I did add five entries in that case and then in two variables, I stored the original size and the original capacity. Size, again how many elements are actually in there, and capacity, how many elements can the collection hold?

A little logic check, after that. I made sure that the capacity was larger than the size by using an expect GE– expect greater than or equal to. I just wanted to make sure that the capacity was larger or equal to the size. And after that, I'm going to reserve 20. I'm going to increase the capacity of my collection by 20 elements. Then again, I store the new size and the new capacity into variables. Then we're going to assert that the sizes–both the original size that I specified earlier, and the new size that I computed after we reserved new capacity– are the same. We don't want the size to increase. However, I want to make sure that the new capacity is larger than the original capacity. And we test that with that assert GE function, an assert greater than or equal to.

You can see the result: the test was run and the test did pass all these assertions and expects. It functioned exactly as we thought: we reserved 20, it increased the capacity up, but it didn't increase the size. The sizes stayed the same. Fantastic.

This was just a simple test just to make sure that our vector collection was working as intended. I wanted to just make sure that add entries function worked. We wanted to add five entries to the collect the vector collection. And then we just assert that the collection size is greater than equal to five. And as you can see, that test was run and it passed perfectly so that add entries function works as intended.

This next one was similar to what we did with reserve somewhat. So we were testing that capacity is greater than or equal to size. So we start with our collection and we tested it at one, five, and 10. We first ensured that the capacity was greater than or equal to the size. Then we added an entry and then we tested again, and then we added four more entries to take it up to five entries that we've added. We tested again, added five more entries, taking it up to 10 that we've added. And each step of the way after those additions, we wanted to make sure that our capacity was greater than or equal to our size. In other

words, the capacity was increasing and this, and it was remaining larger than the size. As you can see, that test was run and that test passed.

So getting into, some meaty topics here, the DevSecOps life cycle. The DevSecOps cycle is a way of structuring development, where we put development and security hand in hand simultaneously just like some of those classic DevOps models that never stop. Once we deploy, we're maintaining rigorous testing in perpetuity. That's exactly the same with DevSecOps. We're just focusing on putting security more at the heart of things. And we want it to be focused from the beginning until after our product is deployed to continuing onward. For that, we have various tools that we can use at certain states for assessing and planning. We can do some threat modeling. There's various software out there for that.

When we're choosing our dependencies, in a previous course I used some OWASP dependency checks to go through third-party dependencies that we might be using and see if there are any weird problems there or anything we can focus on. During our actual development process, when we're actually writing our code, the verifying tests we can do a lot of static code analysis. ParaSoft is the leading man in that realm. It catches a lot of errors and vulnerabilities. We also used CppCheck during parts of this course which can even be integrated into visual studio and run in-IDE, which is very handy. We also want to put our system under stress once we've got things built and we're starting to really integrate things.

And we want to make sure that let's, for example, we're developing a web application. How does that web application work at a hundred users at a thousand users at a 10,000 user level? We can use web load to run automatic load testing and see how our system does. We could deploy that at multiple stages. Again, these are all things that could be automated. These, we could be running stress testing at night, and automatic static code analysis. After, on stuff that we push to collective repositories, we can do penetration testing as an automated thing during our monitor and detect phase using MetaSploit, which is the gold standard for white-hat penetration testing. We can write a wide variety of tests and try to break our security using the many different ways to circumvent all of our measures. That's how we know that we've achieved a good level of security.

Cost-benefit. How much is this going to cost and what are the benefits that we get for that spent cost? Of course, when we're dealing with security, we want to start as early as possible. It needs to be from day one. We need to start with some threat modeling in our design process, so we can help plan code that could be secure as possible. We want to be writing tests right alongside our code. We want to build automated systems that try to break our code at every step to make sure that we're doing the right things. The downside of that is the time cost and specialized information inherent in writing secure code. It does involve a lot more upfront knowledge. And if our staff doesn't have that, we have to look at bringing on either new staff who are more in line with the kind of standards that we're trying to meet, or we need to have some enhanced training. That is going to take our developers away from actually writing code temporarily. However, the time cost is going to pay off. We are going to spend a lot more time upfront writing our automated systems, putting those tools into place that are going to be checking our product at every stage. However, if we get into a potential breach or a code that breaks, or doesn't work as necessary the cost of that far outweighs implementing security. If we're dealing with really sensitive information–we're talking about social security, numbers, addresses, credit card, debit card numbers, all that stuff– we have to ensure that we're doing everything we can to safeguard that information.

So there are a few unanswered questions in this, the security guidelines. So, we talked about what deficient code might look like, but what actually happens if deficient code is found? Is there some type of formal process by which a developer who is found with deficient code sort of logs that and fixes it and keeps a record? Or are we just, if we're, we're just relying on developers to police their code? How are we going to ensure that the guidelines are followed? And second, what is the actual roadmap for doing all this automation? We've talked about a lot of stuff that we can do, but how do we actually achieve it? How do we implement the DevSecOps that we've talked about?

We need to have formalized schedules and a specialized perhaps even team that's in charge of all this automation also, how do we intend to scale security from this guide to all the developer teams? Is it something we're going to do sort of trickle it out a little bit? Are we going to just hold a massive onboarding? Send out an email with a 300-page PDF with all kinds of guidelines so that all the teams can get in the know? It sounds like chaos. So there has to be some sort of formalized method by which we're going to transition all teams to this new security standard also, beyond the secure coding, I would also argue that we need to have a consistent style guide different programmers have different stylistic, little quirks, the classic debate of tabs versus spaces which can cause kind of funky

Undefined stuff to go on in certain code editors again, a small consideration, but definitely something we should look into in the future. And also, what flexibility do we need to have to ensure that secure coding incorporates all types of products? We focus on C++ in this project. What happens to our secure coding principles if a year from now we've got five teams that are working on an iOS app in Kotlin, and we've got another team that's doing some scripting in Python. And another guy is working on a MEAN stack web application. How are we going to standardize and incorporate secure coding into a wide variety of projects that this company might take on?

So going forward I would say that during the hiring process itself, we need to be evaluating the secure coding knowledge of our interviewees by identifying deficiencies and making sure that the interviewee expresses willingness to expand their knowledge in this department. Certainly, we don't want to turn away skilled programmers because they might be lacking in some information that we can teach them on the job. But it's something to take into consideration during hiring in terms of code reviews. Oftentimes we want to make sure and code reviews that the code just does what it needs to do. We're trying to ship a product. Okay. Does it work? Does it achieve the desired function? Great check the box to continue. However, code reviews also need to spend equal time focusing on our security principles. Is this code well written in terms of any vulnerabilities or possible errors that can kind of open the door for security problems down the road?

That goes hand in hand with unit testing. As I said before, 50% writing code, 50% writing tests, that's sort of the universal standard. If we're not writing unit tests, we're not writing code, we're just writing words. Also if we're going to build and automate a continuous integration, continuous delivery pipeline for our existing products that are already there, that we're pushing upgrades to et cetera. We need to have container checking in place. There are tons of tools out there that will look through our Docker images or our YAMLs and just make sure that all those containers are working as they should before we push them into our pipelines. And finally, let's just put the enforcement policies into writing and actionability.

What happens if deficient code is found? What's the formal process? Again, everyone hates to hear the dreaded PIP–personal improvement plan–that Facebook and Netflix have been bandying about over the past few weeks, but there need to be standards and there need to be consequences, especially if we have developers that are struggling to maintain those secure coding standards. It's the responsibility of everybody on the team to ensure that every other member of the team is following the standards. And if there are deficiencies and knowledge, we just need to figure out how to get that person up to scratch, or if it continues, there need to be more policies in place to handle that down the road. But overall, we have a fantastic set of guidelines. We just need to figure out how to enforce those guidelines. What does that look like? Overall, Green pace is going to benefit phenomenally from implementing secure coding procedures. Our reputation is everything in the business in terms of security that is a live-by or die-by kind of reputational hazard. All it takes is one big breach and consumer trust or business-to-business trust quickly evaporates. By adopting secure standards we can make sure that not only are we writing secure code today, but we can continue to innovate and write to share code tomorrow. Thank you for your time.