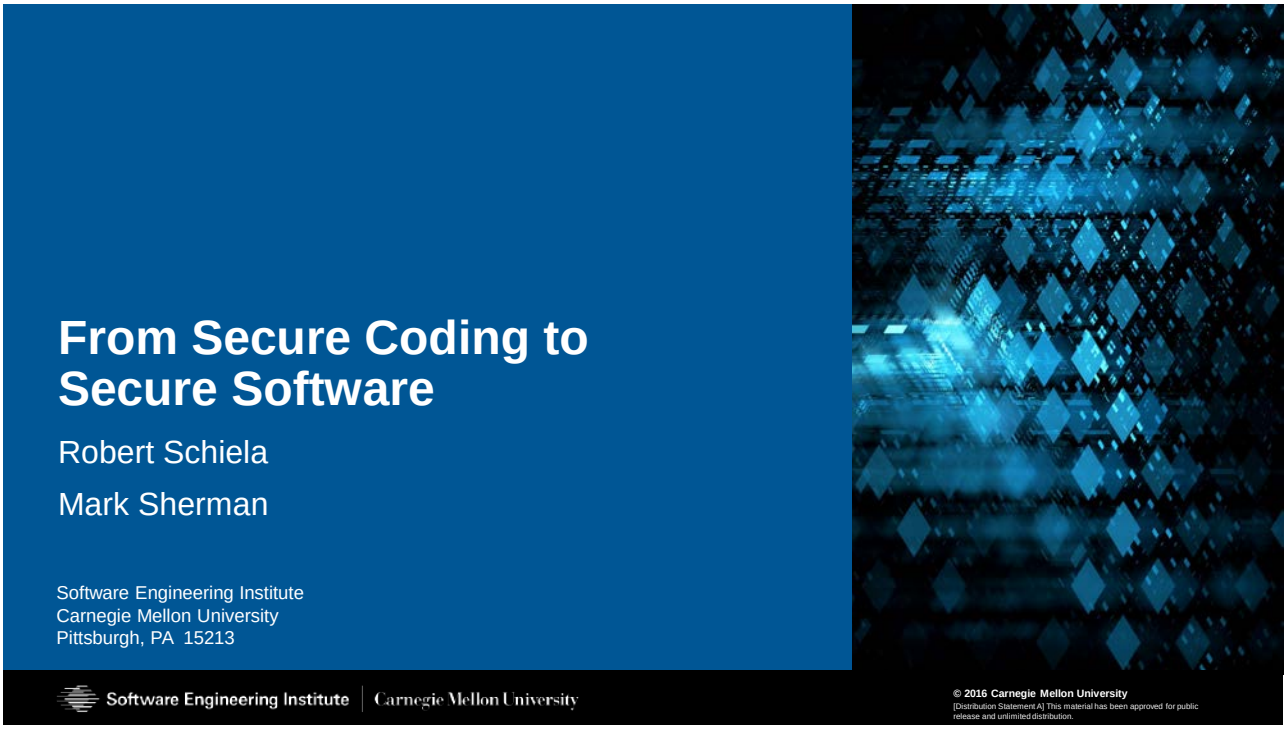
From Secure Coding to Secure Software

Table of Contents



**004 Presenter: And hello from the campus of Carnegie Mellon University in Pittsburgh, Pennsylvania. We welcome you to the Software Engineering Institute's webinar series. Our presentation today is From Secure Coding to Secure Software. Depending on your location, we wish you a good morning, a good afternoon, or good evening.

My name is Shane McGraw, your moderator for the presentation, and I'd like to thank you for attending. We want to make today as interactive as possible, so we will address questions throughout the presentation, and again at the end of the presentation. You can submit those questions to our event staff at

any time by using the Ask a Question tab or the Chat tab on your control panel.

We will also ask a few polling questions throughout the presentation and they will appear as a popup window on your screen. The first question I'd like to ask, and it will appear now, is: How did you hear of today's event?

Another three tabs I'd like to point out are the Download Materials, Twitter, and Survey tabs. The Download Materials tab has a PDF copy of the presentation slides there now, along with other secure coding related work and resources from the SEI. For those of you using Twitter, be sure to follow @cert_division and use the hashtag #seiwebinar. The survey we ask that you fill out upon the completion of today's webinar, as your feedback is always greatly appreciated.

Now I'd like to introduce our presenters for today. Bob Schiela is the technical manager leading the secure coding team in the Cybersecurity Foundations Directorate of the CERT division. Dr. Mark Sherman is the technical director for the Cybersecurity Foundations group, and before coming to CERT, Dr. Sherman was at IBM and various startups. Welcome Bob and Mark. All yours.

Presenter: Thank you very much, Shane. It's a pleasure to be here and hello to everyone out in the web. We're here today to talk about secure software, and in particular--

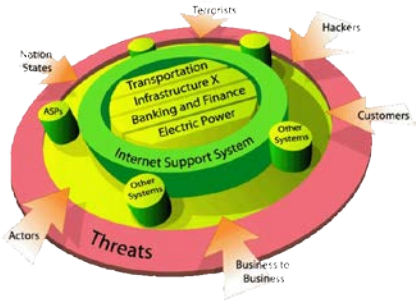
Why Software Security?

Developed nations' economies and defense depend, in large part, on the reliable execution of software

Software is ubiquitous, affecting all aspects of our personal and professional lives.

Software vulnerabilities are equally ubiquitous, jeopardizing:

- personal identities
- intellectual property
- consumer trust
- business services, operations, and continuity
- critical infrastructures & government



**005 --Developing secure software through secure coding practices.

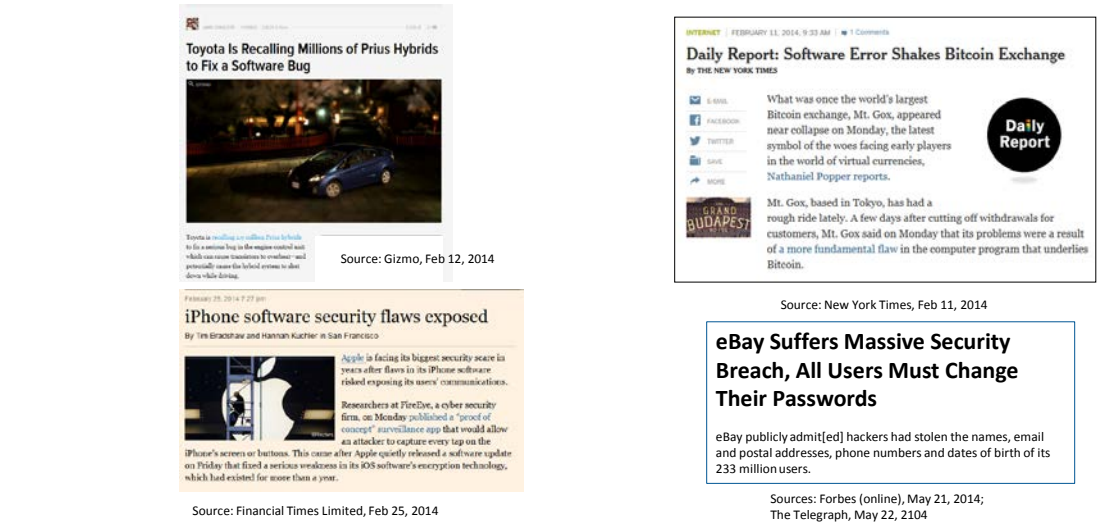
So I want to start off with why is that important. Well, today, more than ever, software has become a part of our lives, integrated into the systems we use every day. More than ever, we're depending on those systems to work. We have cell phones that are embedded in our cars and other devices that we rely on every day, and they're more connected than ever to other systems. Also a lot of these systems are connected in ways that they were never originally designed to be connected, so it's created other avenues for attack.

A great example of all of these conditions-- it was just announced a couple of researchers at DEFCON,

they created a proof of concept ransomware on a thermostat. So IoT devices are definitely vectors for attack and we're starting to rely on them more and more.

So there's a lot of different, important ways that it's affecting our lives through our personal information, intellectual property of organizations being lost, consumer trust, and just general continuity of services. Our economies today, more than ever, and also our nation's defenses are relying on software.

Software and security failures are rampant



**006 So, as we also know, these vulnerabilities are affecting us and we're hearing about them constantly. It seems like every day, or at least every week, there's a new

announcement of somebody or some system being affected by a hacker or an attack. We learn a lot more about the specific coding issues in open-source software, but there are definitely issues with security of software whether it's open source or proprietary.

Software and security failures are expensive

Software and security failures are expensive

Source: New York Times, Jan 10, 2014

Source: Wall Street Journal, Feb 26, 2014

Average cost in a breach:
$158 per record ($221 in US)

Source: Ponemon Institute, "2016 Cost of Data Breach Study: Global Analysis", June 2016

**007 Additionally, the cost of securing the software or, in particular, dealing with the aftermath of a failure, has increased in price because of the amount of data that's being collected and the importance of that data. A recent report in 2016, a couple months ago, the 2016 Cost of Data Breach report estimates that the cost is 158 dollars per record across the globe, and in particular, the data of U.S. is about 221 dollars per record that's lost.

Polling Question 2

Polling Question 2

What programming language are you most concerned about using securely?

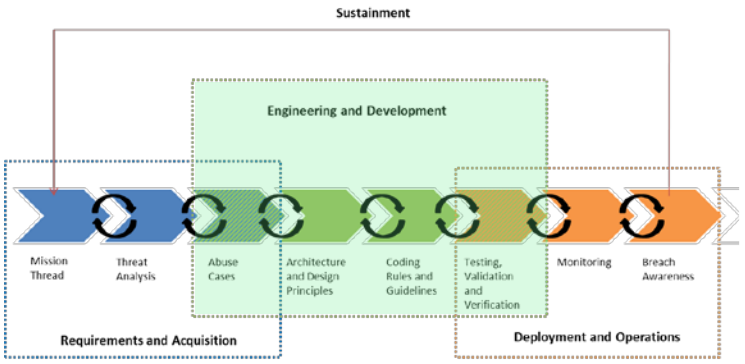
- Ada
- Assembly
- C
- C++
- C#
- Java
- Java-Script
- Objective-C
- Perl
- PHP
- Python
- PL/SQL or SQL
- Ruby
- Swift
- Visual Basic
- Other
- Little to none developed in-house

**008 So we're going to talk a little more in detail about secure coding, which means we're going to be talking about software development, and in particular, developing particular languages. Before we get started in the details, I'm interested in finding out about what languages you're most interested in out there with regard to what you're trying to secure.

Presenter: So the question is posed, so we can see the results in about another minute or so.

Presenter: Yeah, so please go ahead and answer and submit your answer to that question.

Engineering and Development



**009 So, as I mentioned, we're going to talk about secure coding in particular, and here's a standard model of the development lifecycle, and to us we kind of focus on the security aspects of this development lifecycle. The Secure Coding group, and what we're trying to impact, really starts in the green chevron in the center and the one to its immediate right, Coding Rules and Guidelines and Testing, Validation and Verification. That's really what we're trying to focus on improving.

Most Vulnerabilities Are Caused by Programming Errors

Most Vulnerabilities Are Caused by Programming Errors

64% of the vulnerabilities in the NIST National Vulnerability Database due to programming errors

- 51% of those were due to classic errors like buffer overflows, cross-site scripting, injection flaws

Top vulnerabilities include

- Integer overflow
- Buffer overflow
- Uncontrolled Format String
- Missing authentication
- Missing or incorrect authorization
- Reliance on untrusted inputs (aka tainted inputs)

Sources: Heffley/Meunier: Can Source Code Auditing Software Identify Common Vulnerabilities and Be Used to Evaluate Software Security?; cwe.mitre.org/top25 Jan 6, 2015

**010 Why are we doing that?
We're doing that because a majority of vulnerabilities, according to a NIST report, have been found to be related to programming errors, and in particular, a strong majority, if not even more, has been found to be classic errors that are fairly well known, like buffer overflows, cross-site scripting and injection flaws. In particular, top vulnerabilities include integer overflow, buffer overflow, format strings that are not controlled, and then there's also issues with regard to authentication and authorization. And so these are well known, and yet they keep occurring.

Secure Software Development

Secure software development starts with understanding insecure coding practices, and how these may be exploited.

Insecure designs can lead to “intentional errors”, that is, the code is correctly implemented but the resulting software contains a vulnerability.

Secure designs require an understanding of functional and non-functional software requirements.

Secure coding requires an understanding of implementation specifics.

**011 When you're trying to develop software that is secure, it really starts from the beginning, and so we don't lose sight of that, that you need to design it securely. If there are flaws in the design, then it doesn't matter if you use proper coding practices, you're likely to have flaws in the software, and so we keep that in sight as well, that the software has to be designed securely. But then even a strong design can be undone by poor secure coding practices, and we'll talk about that in a few minutes.

Sources of Software Insecurity

Absent or minimal consideration of security during all life cycle phases

Complexity, inadequacy, and change

Incorrect or changing assumptions

Not thinking like an attacker

Flawed specifications & designs

Poor implementation of software interfaces

Unintended, unexpected interactions

- with other components
- with the software's execution environment

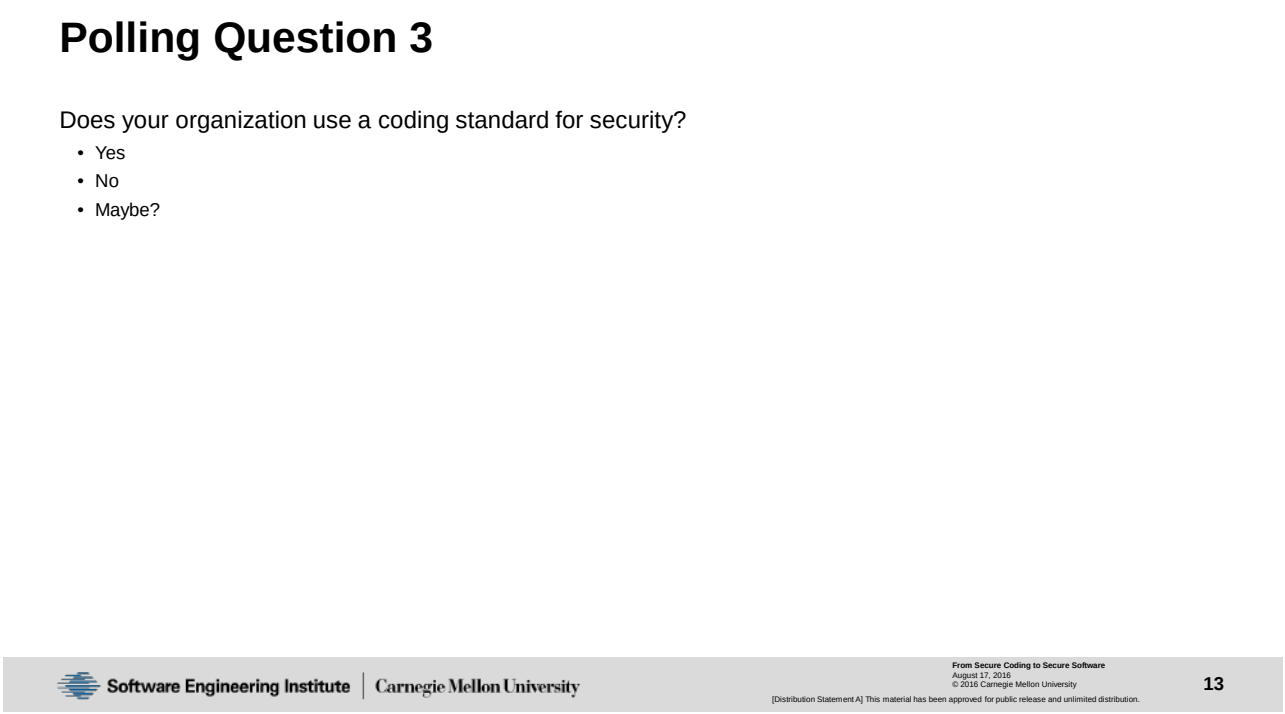
Inadequate knowledge of secure coding practices

**012 So, here's a few common sources of software insecurity, again, kind of across the lifecycle, looking at design all the way through. In particular, just to mention a few of them, the complexity of the systems has grown, the inadequacy of a particular system in the context that it's placed, and change and managing that change across the system, across developers and development teams, is difficult to manage. During the design, not thinking like an attacker, just thinking of use cases of how a user will use it, as opposed to thinking about abuse cases and how attackers might misuse the software to get it to do something the original designers didn't intend is another aspect that can allow you to create software that's not secure.

Let's see another one here. Well, a major one that we'll talk a little bit more about with our roles is unintended and unexpected, generically, interactions, but also unintended, unexpected behaviors. This is, as I was mentioning, systems that have been designed for a particular context or environment or to be used in a particular way, and then later, after it's been developed, is put in a different environment. Or, for unintended, unexpected behaviors, it is writing the software using language constructs that were never intended by the language itself. And so you start violating rules of the language, whether you know it or not, and then the software may do things, after it's compiled and installed on a particular platform, that you didn't intend that leads to security issues.

And finally, one of the things we'll talk about later is inadequate knowledge of secure coding practices, and we'll discuss a little bit today about what some of that knowledge and what secure coding practices mean and how you can gain more knowledge on that.

Polling Question 3



**013 So here we have another question that I would like to ask to the group. Go ahead.

Presenter: So while that one's launching, Bob, I can do some of the results of the last one.

Presenter: Sure, sure.

Presenter: So it looks like we had 24 percent with Java. It looks like C-hashtag or C-pound was--

Presenter: C Sharp.

Presenter: C Sharp is at 20 percent, then C++ 18. Those were the three main winners.

Presenter: Okay. Great, great, great. Thanks for that. And actually,

before we go to the polling question, unless you already pushed it, do we have any questions from the audience up till now?

Presenter: We actually have one interesting comment, which we'll actually talk about. It was a question about that SQL injection was noticed quite a while ago and yet we're still hearing about it, and I'll paraphrase: Why is that? And we'll actually talk a little bit about in a moment.

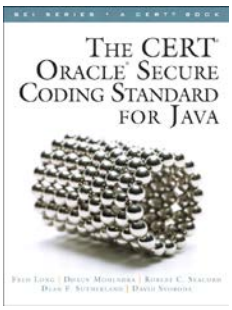
Presenter: Bobby Tables. I love Bobby Tables. It reminds me of me because my name's Bobby, so. Yes. Yeah. That goes to the design issues and some things that have nothing to do with the coding, or in one language that then is interpreted by a different language.

Presenter: And just to wrap up our results, 47 percent with yes, they use a coding standard for security, 23 percent no, 29 percent maybe.

Presenter: Okay, great. Thank you.

Coding rules – 2016 Edition

Coding rules – 2016 Edition



- Collected wisdom of programmers and tools vendors
  - Fed by community wiki started in Spring 2006
  - Over 1,500 registered contributors
- C Coding Standards
  Available as downloadable report

http://cert.org/secure-coding/products-services/secure-coding-download.cfm

- Java Coding Standards
  Available as book
- C++, Perl, and "Current Standards"
  Available on Secure Coding Wiki

https://www.securecoding.cert.org/

**014 And so what are the standards? I've mentioned them several times here. So at CERT, in the Software Engineering Institute, we started codifying best practices for coding securely in specific languages about a decade ago, and it all started with the recognition that a lot of these issues are common. So we started looking at what's common and how can we prevent these common errors. In particular, which errors are being caused by misuse of the language. And so that started with a general best practices book, and then as our knowledge matured, we started writing specific coding standards for C and for Java eventually. And so we're going to take a look talk a little bit more about what's in those standards, but they're generally a compilation of how to use

the languages and the constructs in the language securely, avoiding common flaws.

And so recently, we just released-- a couple months ago we released a new version of the C coding standards as a downloadable report. The link is there so you can download that freely. The Java coding standard is available as a book, and we also have C++ and Perl standards that are in development but available on our secure coding wiki, as well as what I call, or we call, the current standards. They're kind of the in-flight, in-development beta version of the standards that are available on the wiki, but they have not been-- the changes since the last iteration of the publication, basically. And so if you need a snapshot of it, you can download the PDF, but if you want the latest and greatest rules, and I'll even mention a couple things that have changed in the last couple months, you can get them from the wiki. Now, are there any other questions?

Presenter: We do. Just one from Ed asking, "Any work on secure Python programming?"

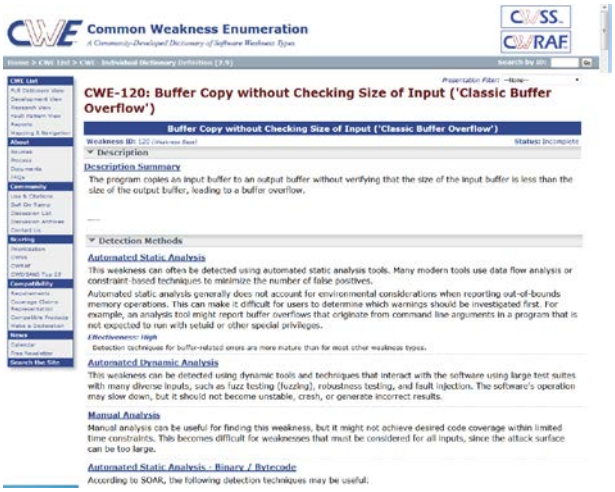
Presenter: Sure. So we have considered working on a couple-- developing languages on a couple other-- sorry-- standards on a couple other languages, Python being one. Another common one, as you mentioned, the second option was C Sharp. We don't have one for that. And several customers recently have been asking us about Ada, which is

kind of a surprise, but it's come up. We're still kind of deciding what we're going to develop. We might start developing, as I said, kind of a beta version on the wiki soon, but that kind of depends and is often driven by customer demand-- customer often meaning some sort of funding source. So it just depends on the demand.

And actually, with that, I was wondering, Mark, if you could tell the audience a little more about the rules and how we develop them.

Presenter: Sure, I'd be happy to, and we'll see some of the history and motivation as to why things don't seem to go away, and some of the discussion that has been going on in the chat address some of the issues, which is the reason why other languages that you might use to prevent them aren't widely adopted, is because they make tradeoffs in time and space that programmers may want to use, but that also makes it a challenge in adopting the rules. So let's consider one particular example.

CWE Guidance



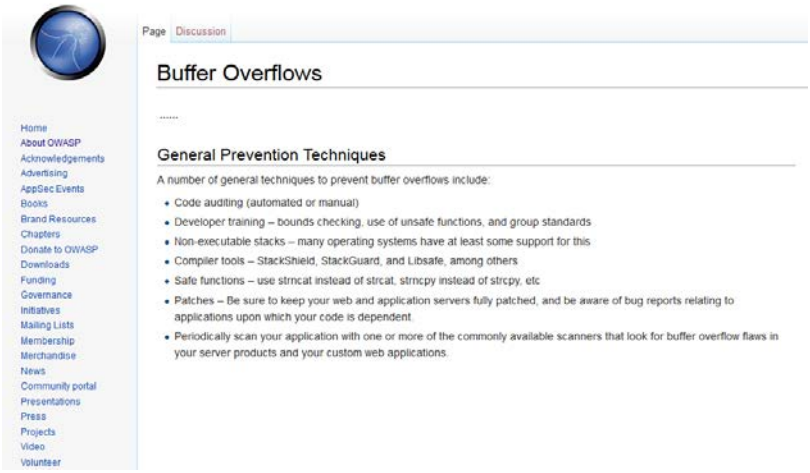
**015 SQL injection was the one mentioned in the chat room, saying, "We know about this. How come it's still going on?" And SQL injection actually is an example of a more broad class of problems, which I call the eval problems-- basically taking in a string of some sort and then performing an evaluation on it-- eval if you're a LISP guy; there's exec if you're a SQL guy; there's a variety of different verbs for that, but the idea is you're taking a piece of code, you trust it, you basically say, "Go execute this code," and it's bad code and you get something like SQL injection.

And part of the challenge in dealing with that is being able to give good advice to the developer on what they should do about it, and rather than

take SQL injection, or cross-site scripting, which are a little more complicated examples, I'm just going to walk you through a little bit of a more simple example, buffer overflow. Buffer overflow, in case you've been living under a rock, is an example where you're either reading from a space that you shouldn't be or writing to space that you shouldn't be-- a mismatch in two buffers usually-- the size of two buffers.

And so there's a lot of talk, a lot of guidance about what to do here, and just to show you sort of the two most common sources of guidance that people look to, CWE that MITRE sponsors-- or MITRE hosts, I should say-- I think DHS actually sponsors it-- Common Weakness Enumeration. This is a particular page, and the one I picked is their guidance for classic buffer overflow, and I don't know if you can read it on the fly here, but you'll certainly be able to study it later, it says, "What should you do about buffer overflow?" It says, "Well, read your code very carefully." It says, "Have someone else read your code very carefully." It says, "Run a tool to see whether it can find the problem or not." That's not very specific advice that you can give to the programmer on what to do. The programmer, when they write code, doesn't intentionally write bad code.

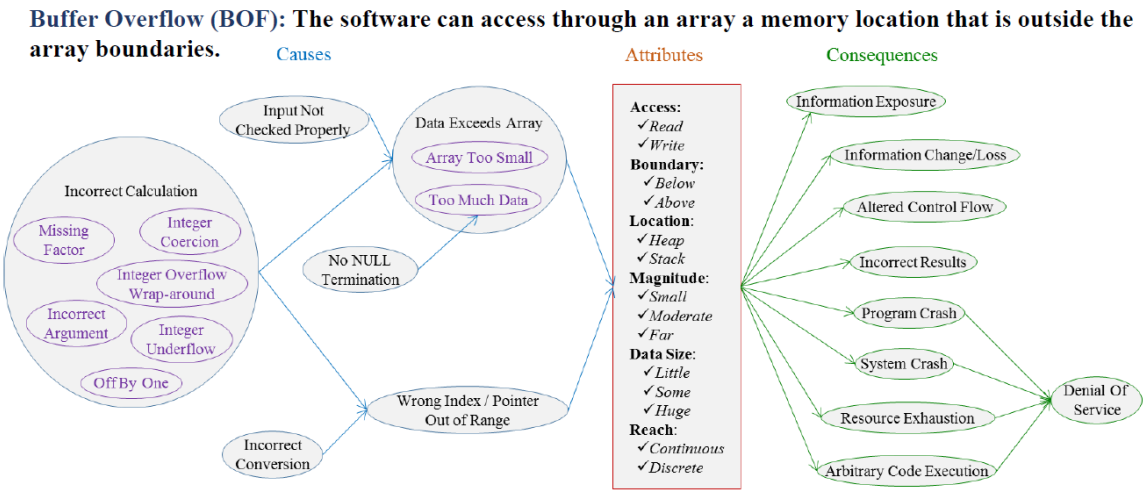
OWASP Guidance



**016 Probably the next most widely used source of guidance is OWASP, the Open Web Guidelines, and again, here's the generic buffer overflow guidance that they give, and they say, "So, what should you do to prevent it?" Well, it says, "Code carefully." It says, "Read your code. Have developers look it over. Run some tools on it." Sometimes you might even see things like, "Check that your indexes are okay." That's a little better, but still is not very prescriptive of what you should do, or proscriptive of what you should avoid. And again, let's consider some reasons why.

Buffer overflow has many causes

Buffer overflow has many causes



Source: Bojanova, et al, "The Bugs Framework (BF): A Structured, Integrated Framework to Express Software Bugs",2016, http://www.mys5.org/Proceedings/2016/Posters/2016-S5-Posters_Wu.pdf

**017 Like, again, the usual buffer overflow. It turns out buffer overflow-- and this is a simple one, as I said. For the other ones, they're much more complicated. Buffer overflow actually has many, many different things that cause that problem to occur. So, for example, it could be that the input that came in through the system is tainted. It could be that you didn't have the right scaling factor. It could be that you had some sort of integer overflow, or integer underflow. Now, that one may not seem apparent, but let me give you an example of how that might work in practice. And again, it goes back to the realization that people pointed out, that when you're writing code, especially if writing something like C, you're very conscious about trying to be very

fast, trying to be very small. So even if you give the guidance, check your bounds, and that will prevent buffer overflow.

What does a C programmer do? Well, the usual calculation in C for array access, which I suspect most of the folks on this webinar given the background that they have, is basically taking an address, a pointer, and you add a number to it, and then that gives you a pointer into the array, and then you use that to index. And so the efficient way in C in order to see whether you've exceeded the bounds is you start with the start of the array, the pointer, you add in what you think is the test index, some value that you read in from the outside, and you compare it to the pointer that is at the very end. You think you've now checked the bounds, the thought being-- let's say if you have an array of 10 and you get an 11, it's a little bit beyond the bounds. You see that it's beyond the bounds. You say, "Oops, there's an error." The problem is, if you're having something malicious, is that it's not 10 going to 11 that comes in, it's 10 going to 100 million that comes in. When you add a gigantic number to that initial address, you get an integer overflow. That's a fancy way of saying the number wraps around, and actually will be below the original base. So if you want to see have I gone beyond the end of the buffer, the answer is no. It'll look like you're in front of the buffer and you'll say, "Everything is fine," and then you start getting into problems.

Now, there is a way to do this kind of buffer overflow, checking, bounds checking, but it's a lot more subtle than what you might expect to do as a standard C programmer. Coming up with those kind of actionable, precise rules that actually help you build programs that are resilient is really what rules ought to be about, as opposed to what we see here in a lot of the overall guidance, which is useful at one level but not really actionable.

Learning from rules and recommendations

Learning from rules and recommendations

Rules and recommendations in the secure coding standards focus to improve behavior

**018 And so what a couple of groups have done-- we are one of them. Frankly, MISRA is another one-- is develop standards that don't say, "Don't do buffer overflow." They say, "Here's how you should carry out the pointer arithmetic in these kinds of situations so you don't

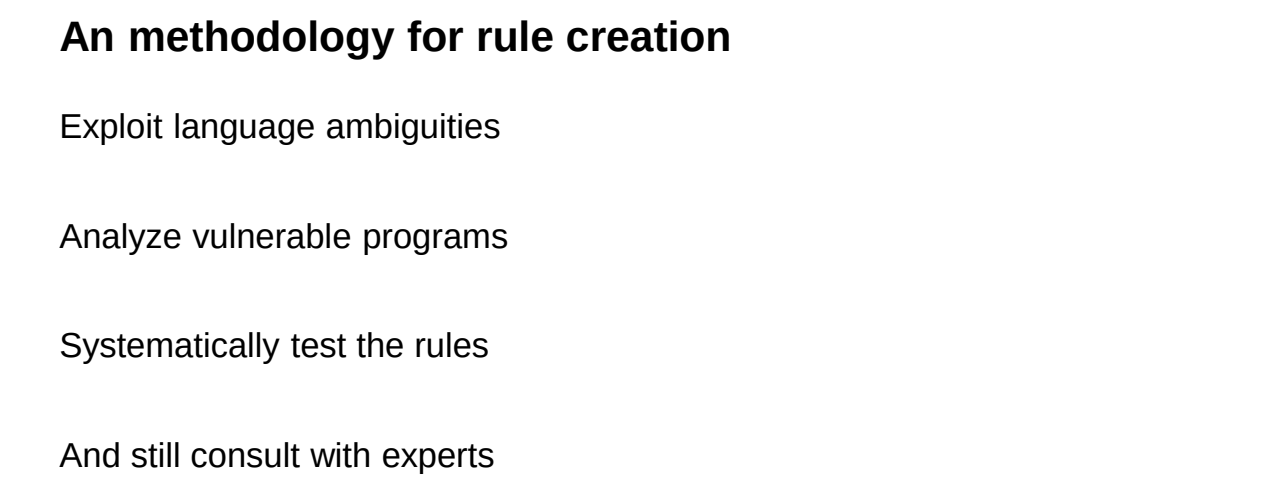
generate these overflows, and you actually check to see whether you're in bounds or not." That is a way then that you can make sure that you are meeting the goal of not creating a buffer overflow from at least one source, the integer overflow.

Now, in the case of both us and MISRA, the way we do it is we have some rules-- that's what the very top part of this is-- but perhaps as important is in the next section of the standard, we give examples of what code that looks good might seem, but really isn't, as the example I gave where you're trying to do the pointer calculation and you wind up wrapping around and you think that you haven't exceeded the end of the buffer, but you have. Showing you then how you might have written it, which is not correct, and then showing you the right way to do it is provided so you can see the better way to do it and actually accomplish what you want, to not have the buffer overflow.

So I mentioned CWEs don't have as many of these kinds of things. We focus on it. MISRA focuses on it. We focus on particular elements to make things secure. MISRA focuses on particular things for safety in embedded systems. Just to give you an example, we worry about things like integer overflows driving buffer overflows. They worry about running out of space. So their rules, for example, would include, "Don't use malloc. No recursion in functions," things of that sort, as a way to

maintain static memory allocation so that your card doesn't run out of space as it's trying to do some kind of thing. But the point is, these two sets of rules are actionable as opposed to many other sets of rules, which are, "Don't do something bad." And for something like SQL injection, it's a lot more involved to explain to a developer how it is that you should code in order to prevent SQL injection, other than something very overwhelming like, "Don't do any executable code." Which is kind of difficult in today's environment where you want maximum flexibility.

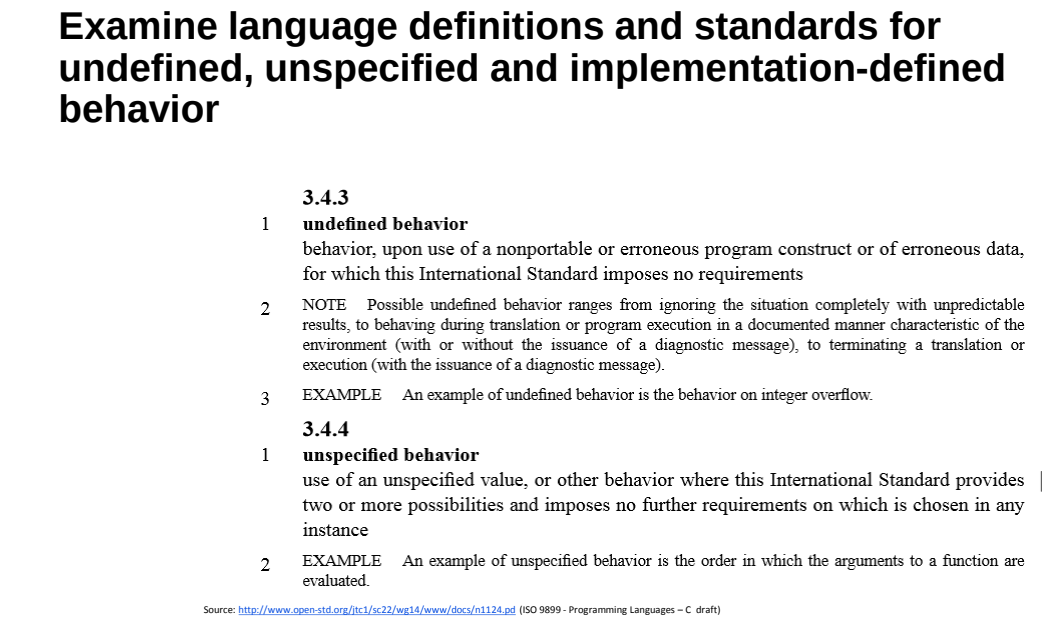
An methodology for rule creation



**019 So how do we generate these kinds of rules? Well, we actually have developed a methodology. So we've done this for a couple of

languages and we can apply these to other languages, and if you wanted to build your own set of rules for your own company, you might do this as well. First, we look for ways to exploit language ambiguities. We then actually look at vulnerable programs. Where have things gone wrong? We create some rules and systematically test them, and we then ask for personal opinions from experts. Most of the other rules that we talk about that you read are frankly just that last one. A group of very smart people get together and say, "I think it looks like this," but then you wind up with these sort of generic rules of what can you do about it.

Examine language definitions and standards for undefined, unspecified and implementation-defined behavior



Examine language definitions and standards for undefined, unspecified and implementation-defined behavior

3.4.3

1 **undefined behavior**
behavior, upon use of a nonportable or erroneous program construct or of erroneous data, for which this International Standard imposes no requirements

2 NOTE Possible undefined behavior ranges from ignoring the situation completely with unpredictable results, to behaving during translation or program execution in a documented manner characteristic of the environment (with or without the issuance of a diagnostic message), to terminating a translation or execution (with the issuance of a diagnostic message).

3 EXAMPLE An example of undefined behavior is the behavior on integer overflow.

3.4.4

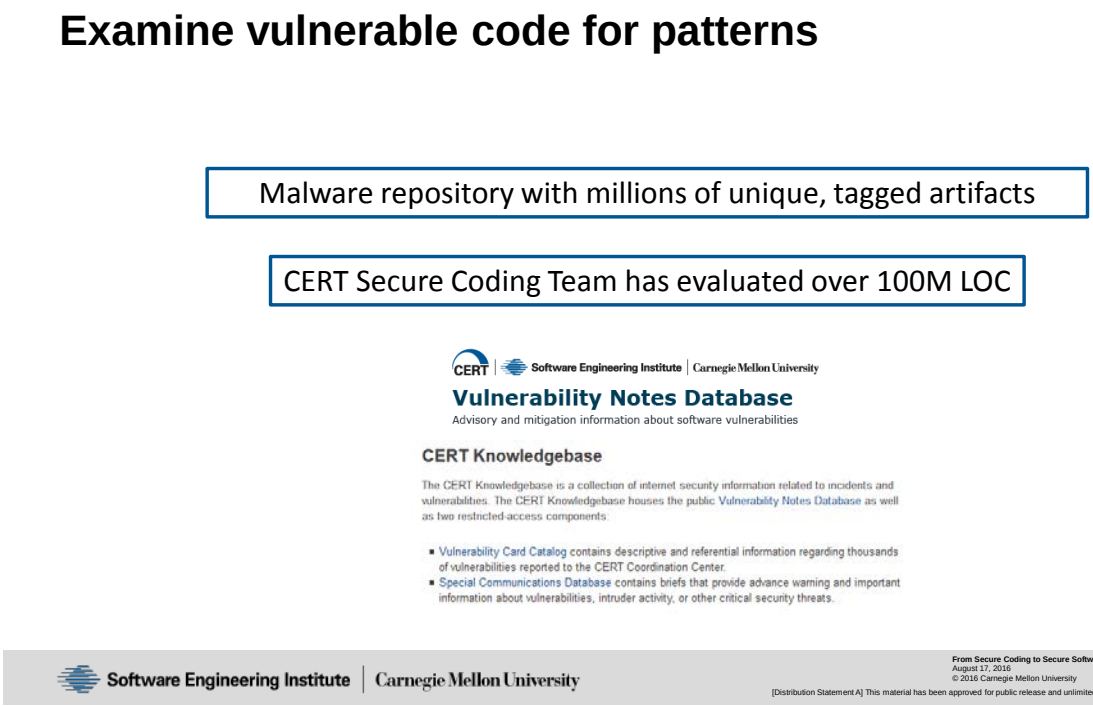
1 **unspecified behavior**
use of an unspecified value, or other behavior where this International Standard provides two or more possibilities and imposes no further requirements on which is chosen in any instance

2 EXAMPLE An example of unspecified behavior is the order in which the arguments to a function are evaluated.

Source: http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pd (ISO 9899 - Programming Languages – C draft)

**020 So, as an example, this is from C. The C language has things like undefined behavior, unspecified behavior, implementation-defined behavior. Each of those ambiguities in the language offers an opportunity for exploitation. Now, we know why they were put in the language. They were put in because different compiler writers for different architectures and different systems wanted to have different interpretations of those so that they could optimize the language for their system. At the same time, that leaves open the other interpretations which can be exploited sometimes for security flaws.

Examine vulnerable code for patterns



**021 The second thing we do is we look at a lot of existing programs.

We're fortunate in this respect at CERT that we have a huge malware repository, and as part of our research work, we have evaluated a lot of code, over 100 million lines of code, and so we have seen both other people's code and problems that have arisen in the security area, and basically how the bad programming happened, and so that gives us the patterns of things to look for to at least advise people of what to avoid.

Implement candidate rules and run against sample code

Implement candidate rules and run against sample code

- Focus rule when possible to
  - maximize true positive of weakness (tag bad code)
  - minimize false negative of weakness (don't tag good code)
- Write program to evaluate source code for particular rule
- Run program against collection of known bad source code and a collection of other (suspected good) code to check sensitivity and specificity of results

**022 Having done that, we then take the next step, which is kind of tricky as well...we then kind of come up with rules. If you've used any of the static checkers, and I suspect many of the people have, the problem of false-positives is endemic.

You put something in and you get hundreds and hundreds if you're lucky, thousands and thousands if you're not, of messages telling what's the problem with your code, and it's very difficult to actually then do something with that. And so the challenge is coming up with a rule that will maximize the true positives--you really want to find the bad code--but also not tag the good code. So what we do is we generate some of these rules, and because we want them to be precise and concrete, we write a program that implements them and we run them over a lot of code, usually around tens of millions of lines of code, and we see what the results are, and then sometimes we tweak the answers and then get something which is a lot more precise.

Experience with systematic testing

Experience with systematic testing

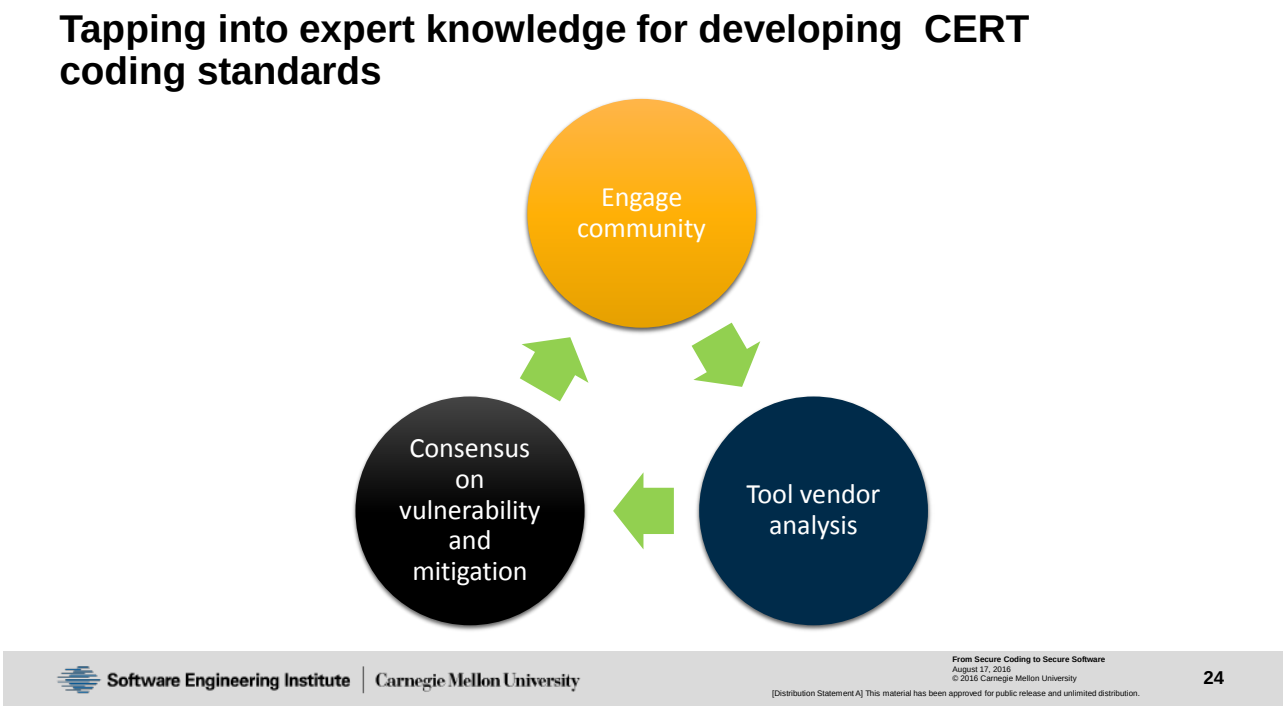
- Candidate rule typical evaluation
  - 10 iterations of proposed rule and associated checker
    - 7 internal evaluations
    - 3 external evaluations

- Each evaluation iteration carried out against > 10M lines of representative code
  - Variety of domains
  - Variety of code quality

- As part of creating C++ standard, general methodology applied to generate 46 rules and corresponding Clang C++ checkers
  - 19 by CERT researchers, 27 by others

**023 For example, we typically do this ten times in order for us to get the rule right, if you're curious about our experiences here, and as part of the most recent standard that we're working on in C++, in about a year's time we were able to, working with others, generate about 46 rules. So if you want to know how much effort is involved in trying to put something like this together, that's been our history here.

Tapping into expert knowledge for developing CERT coding s tandards



**024 And of course, while we do the systematic analysis, judgment always comes into play as well, and so we do have a wiki that we invite people, experts and other practitioners, to participate in, and we'll probably talk a little bit about that more later, but it's a way for

both people who want advice to see the discussions of the various kinds of rules and their implications, and for those who have advice to offer, a forum by which they can share that knowledge, and of course it feeds into our process for how we go about developing these new rules. And to give you some more concrete examples, I'm going to turn this back over to Bob and let him show you some of the rules.

New Rule Example

New Rule Example

EXP46-C – Do not use a bitwise operator with a Boolean-like operand

```
if (!(getuid() & geteuid() == 0)) {
  /* ... */
}
```

```
if (!(getuid() && geteuid() == 0)) {
  /* ... */
}
```

CWE-480, Use of incorrect operator

**025 Presenter: Thank you, Mark. Before I get started with that, let me just say that I've been watching some of the comments and there's a really great discussion going on. Thanks to everyone for keeping it professional. I'd love to throw in some comments on everything that's

been talked about. There's one in particular though that I really want to mention, or refer to, a question by Brian about, "Aren't the C++ rules and recommendations marked basically with page warnings right now about them being outdated?" We're actively updating the C++ wiki site right now, and as part of that effort, one of the first things we did was went through and marked mostly recommendations-- not rules, but most of the recommendations. We just noticed that a lot of them have not been reviewed in a while, and when we reviewed them we had some concern about some of the accuracy, and since we had to stage our efforts to update the C++ guidelines, we decided to start with the rules because those are the most important, and so we're working on those first. So you shouldn't really see many, if any, of the outdated marks on rules because we're updating those now, and then once we get to the recommendations we'll peel off those warnings about the outdated nature of them, and you're welcome, Brian.

So as Mark said, I'm going to just mention a couple of examples that I pulled from the standard. They're relatively simple because we don't have a lot of time to go into in-depth code, but these are two examples that I pulled out specifically that are in the updated C standard in the PDF, compared to the previous version that was published in late 2013, early 2014.

So this first one is kind of a simple piece of code, and here is the rule: Do not use a bitwise operator with Boolean-like operand. Depending on its use-- I mean, it's a conditional, so it might be used and might go down a path that you're not intending, which might lead to a security issue, but it looks pretty innocuous. You see an "and" there and you see a comparison with an "equal to". The issue here is that that's a bitwise "and", not a logical "and", and it's a fairly common issue, and a lot of static analyzers will find this, and so we added this as a rule recently, and just to show you the comparison, that's the fix there with the logical "and", and this maps to CWE-480 for anybody looking against the CWEs, as Mark was talking about, of use of incorrect operator. And so this is a rule that we have added within the last couple years.

Updated Rule Example

ARR38-C – Guarantee that library functions do not form invalid pointers

```
if (1 + 2 + payload + 16 > s->s3->rrec.length)
   return 0; /* Silently discard per RFC 6520 */
```

Heartbleed.com

CWE-119, Improper Restriction of Operations within the Bounds of a Memory Buffer

CWE-121, Stack-based Buffer Overflow

CWE-123, Write-what-where Condition

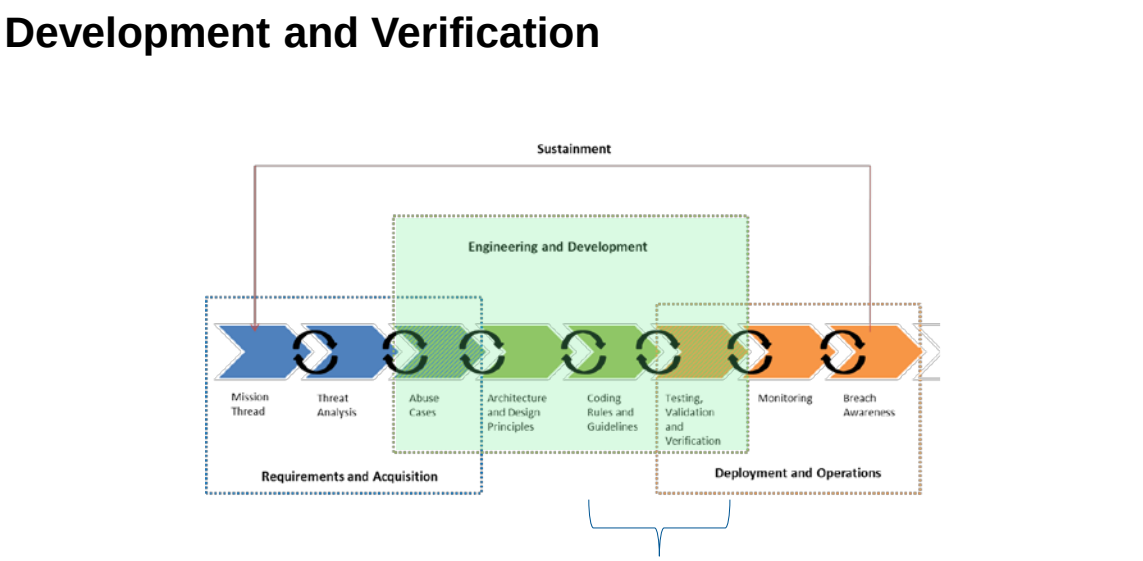
CWE-125, Out-of-bounds Read

CWE-805, Buffer Access with Incorrect Length Value

**026 Another one that I have here, more so because of its popularity or notoriety, is related to the Heartbleed issue of 2014. So this is not a new rule, but we've also updated the rules in the comments and notes to keep them current when examples pop up that become fairly well known. And so here we see-- this is the correction of the code. The issue, as many people know, was that Heartbleed was an SSL defect where it was-- you could query it with a heartbeat request, and it would give you more memory than it should have back as a response, and it was basically an information leakage issue, and the reason was because they didn't have this simple comparison here. What it was doing was asking the user, or the code-- it allowed a user to ask for a particular size and it didn't check to

make sure that the size it was asking for actually matched the size of the buffer that it was supposed to respond with. And so you could ask for a buffer larger than you should have been able to get access to and it would give you information, such as SSL keys and other information that you shouldn't have access to, and all it required was this simple comparison, making sure that the payload that was asked-- the size that it was asked for actually met the size of the buffer. And so that, again, is not a new rule but we added a reference to this particular piece of code, and this rule actually is a very long rule. There are lots of examples and lots of nuances, but it also, as you can see, is pretty connected to-- or refers to several CWEs.

Development and Verification



**027 So now we're going to talk for a couple minutes about how can you use these rules, and in particular secure coding, and adopt.

So first, again, just to kind of focus on where we are, here's our secure development lifecycle, or software development lifecycle, and focusing on coding rules and guidelines and testing, validation and verification, and largely for secure coding, development and verification is really where the action happens. So you need to know how to develop secure code, and then you need to either be able to analyze it, review it, test it, or know what secure code looks like to verify that those practices and those coding constructs were used.

DISA STIG Requirements

DISA STIG Requirements

Application Security STIG Requirements:

- APP3550: CAT I – not vulnerable to integer arithmetic issues
- APP3560: CAT I – does not contain format string vulnerabilities
- APP3570: CAT I – does not allow command injection
- APP3590.1: CAT I – does not have buffer overflows
- APP3590.2: CAT I – does not use functions known to be vulnerable to buffer overflows
- APP2060.1: CAT II – development team follows a set of coding standards
- APP2060.2: CAT II – development team creates a list of unsafe functions to avoid and include in coding standards
- APP2120.3: CAT II – developers are provided with training on secure design and coding practices on at least an annual basis

From Defense Information Systems Agency Application Security and Development Security Technical Implementation Guide, V3 R10 (2015)

**028 For the people on that are in the government or, in particular, in defense that have to deal with DISA STIGs-- that's the Defense Information System Agency Security Technical Implementation Guide-- I just have this as a reference. These are just some of the STIG requirements that are related to our secure coding rules and standards. So if you're trying to-- and you notice there's several CAT I's and a couple CAT II's-- you're trying to-- if you're required to address these, then, following the standards or setting up your own standards based on our standards will help to effect that.

And you'll notice the last-- or I'll point out the last one there is that developers are provided with training on secure design and coding practices, and we'll come back to that.

Adopting Secure Coding Practices

Secure Coding Infrastructure
- Defining Secure Coding Practices
- Influencing Language Standards
- Influencing Tool Vendors

Processes
- Coding Standards and Security Standards, Testing

Technology
- Tools: IDE's and Analyzers
- Automated transformation and remediation

People
- Workforce Development

**029 So, adopting secure coding practices, how to do that. Well, we're trying at CERT, at the Software Engineering Institute-- we're trying to do a few things to help the community, and so I wanted to start with that just for a moment. It's what I call secure coding infrastructure, or community adoption.

So as I mentioned, we're obviously defining secure coding practices. We're also, to help the community adopt them, we're trying to influence language standards, and if we get a chance, I know there was some discussion on the chat about different languages, and maybe we'll have a moment to talk about the languages, although it seems like other commenters have addressed some of that. But we're trying to influence

language standards so that they adopt secure code, improve the language itself, to make it easier for developers to develop secure coding. And we're also trying to influence the tool vendors. The tool vendors are also connected to the language community, so it's not too far, but trying to help the tool vendors, and/or help people's awareness of the tools that can help their secure coding.

So we'll talk a little more about processes, coding standards, technology that you can use, and workforce development and training.

Risk Assessment

Risk Assessment

Risk assessment is performed using failure mode, effects, and criticality analysis.

Severity—How serious are the consequences of the rule being ignored?

Value	Meaning	Examples of Vulnerability
1	low	denial-of-service attack, abnormal termination
2	medium	data integrity violation, unintentional information disclosure
3	high	run arbitrary code

Likelihood—How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?

Value	Meaning
1	unlikely
2	probable
3	likely

Cost—The cost of mitigating the vulnerability.

Value	Meaning	Detection	Correction
1	high	manual	manual
2	medium	automatic	manual
3	low	automatic	automatic

**030 So, one of the issues with adopting this and improving your practice-- well, I'll start-- and this has been commented in here-- I'll talk for

a couple minutes about developing your own standards. So developing your own secure coding standards with something to base off like ours is a really good start. The problem is it may seem overwhelming at first because the secure coding standards are fairly voluminous. Right now I think there are 99 C rules that you should be following.

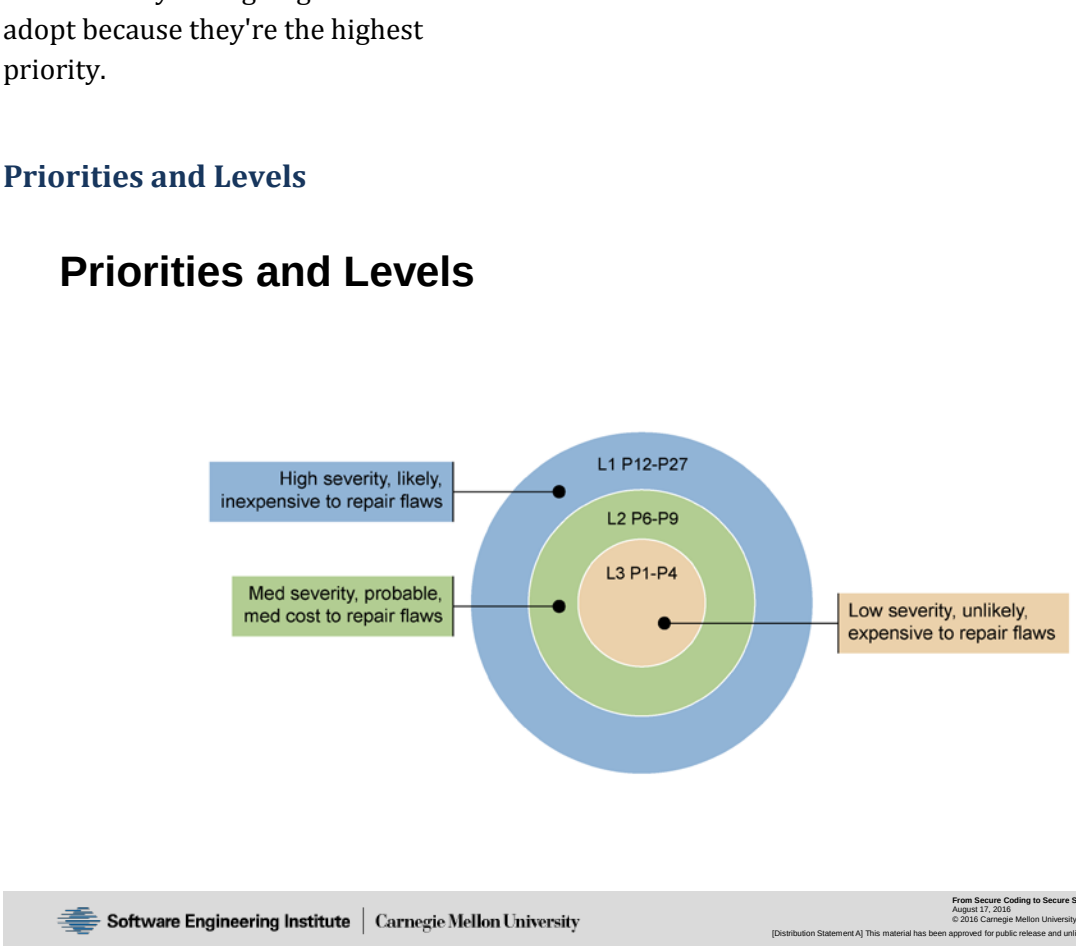
So one of the challenges is to prioritize those, especially when you're trying to adopt them. And so we have a few mechanisms of guidance of how to prioritize. One is we have a risk assessment, and so the risk assessment, for each role, provides a value of how risky a defect or weakness related to that rule is such that it's likely to end up as an exploitable vulnerability.

And so here we have this rating and it's on three different dimensions-- severity, how bad would it be and how exploitable, or what would the effect be, the worse being running arbitrary code and giving control up; the likelihood being another dimension; and then the cost of remediating or mitigating, which is how hard is it to find and/or fix, and as Mark mentioned, a lot of tools give out a lot of false-positives, so finding can be a challenge, or as much of a challenge, if not more, than fixing it once you find the issue.

And so you can use our risk assessment on these rules to decide

which rules you're going to first adopt because they're the highest priority.

Priorities and Levels



**031 And so we mapped those into these priority levels just to make it a little easier, rather than having 27-- and the actual numbers are a little less-- but there's three different ranges for the priority levels of rules.

Conformance Testing

The use of secure coding standards defines a proscriptive set of rules and recommendations to which the source code can be evaluated for compliance.

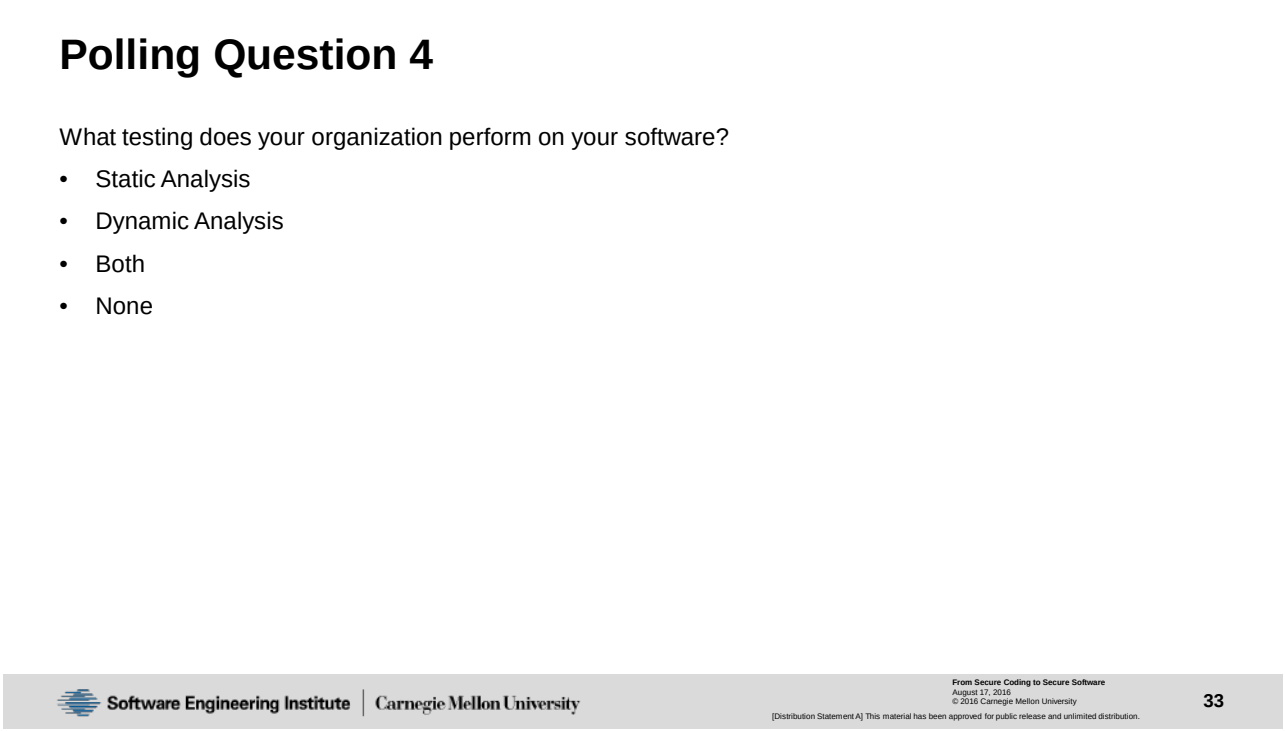
For each secure coding standard, the source code is certified as provably nonconforming, conforming, or provably conforming against each guideline in the standard:

Provably nonconforming	The code is provably nonconforming if one or more violations of a rule are discovered for which no deviation has been allowed.
Conforming	The code is conforming if no violations of a rule can be identified.
Provably conforming	Finally, the code is provably conforming if the code has been verified to adhere to the rule in all possible cases.

Evaluation violations of a particular rule ends when a “provably nonconforming” violation is discovered.

**032 What you also want to be trying to do ideally is looking for code that conforms to the secure coding standards, or the secure coding standards that you choose to adopt, and so that's kind of the goal of analyzing and verifying your code, is you're looking for conformance because conformance means that the software will be much more secure because it won't have these weaknesses that lead to vulnerabilities.

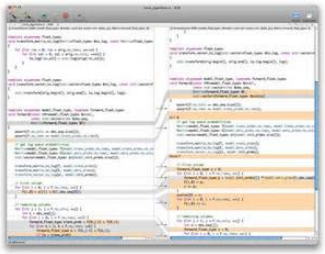
Polling Question 4

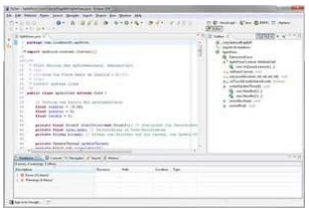


**033 With that-- and before we get started talking about tools and analysis-- and I know there's already been some comments on the chat about that-- I was wondering if we could find out more about what you use to analyze currently in your processes.

Presenter: So that question is posed and we can wait for some results and move on, Bob.

Tools encourage application of secure coding





Moving rules into IDEs improves application of secure coding:

- Early feedback corrects errors on introduction.
- Exceptions are understood in context.

Adoption of secure coding IDEs

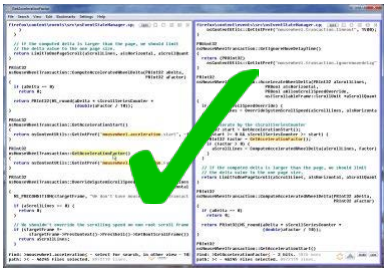
- help deploy tools
- training on tools
- extend tools to meet targeted needs

**034 Presenter: Sure, sure. And so now to talk about tools, the first tool that a lot of people don't think about is the IDE itself, the environment that you're developing. Those tools can provide warnings that can be really helpful to, at first sight, find and fix some simple issues, and so we absolutely recommend looking at those diagnostics from the IDEs. IDEs are getting smarter about presenting information to the developers, and so what we're trying to do in some of our research is align our diagnostics, or diagnostics that tools find related to secure coding vulnerabilities, and present that to the developer while they're coding, because finding and fixing them, of course, earlier in the lifecycle is cheaper, but finding them right when they write them is the

cheapest it can be because they're already presently-- mind is present in the code they're writing.

Static Testing – Source code analysis tools

Static Testing – Source code analysis tools



Secure Code Analysis Laboratory (SCALe)

- C, C++, Java, PERL, Python, Android rule conformance checking
- Thread safety analysis
- Information flows across Android applications
- Operating system call flows

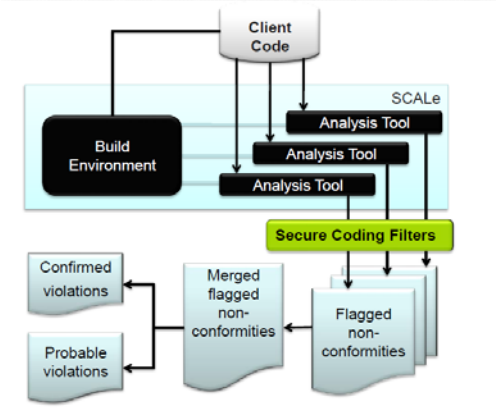
**035 Static analysis is definitely a practice that should be used. There was a question about one particular tool, whether or not it's good enough. We have a recommendation that no tool has complete coverage. Every tool has prioritized or optimized particular types of defects that it's trying to find with really high accuracy, to try and reduce the amount of noise of false-positives. And so that's where their market differentiation is, is what they're trying to find and help you fix. So we recommend that you use multiple tools, and as I said, consider your IDE, or your development

environment tool, or even if you're programming in GCC, just running it with diagnostics.

Similarly-- and Mark mentioned this-- there was also some discussion in there about how different languages are optimized for different things, some for speed, some for safety and protection. The static analysis tools and even the dynamic analysis tools are optimized for different things.

SCALe Multitool evaluation

SCALe Multitool evaluation



Improve expert review productivity by focusing on high priority violations

Filter select secure coding rule violations

- Eliminate irrelevant diagnostics
- Convert to common CERT Secure Coding rule labeling

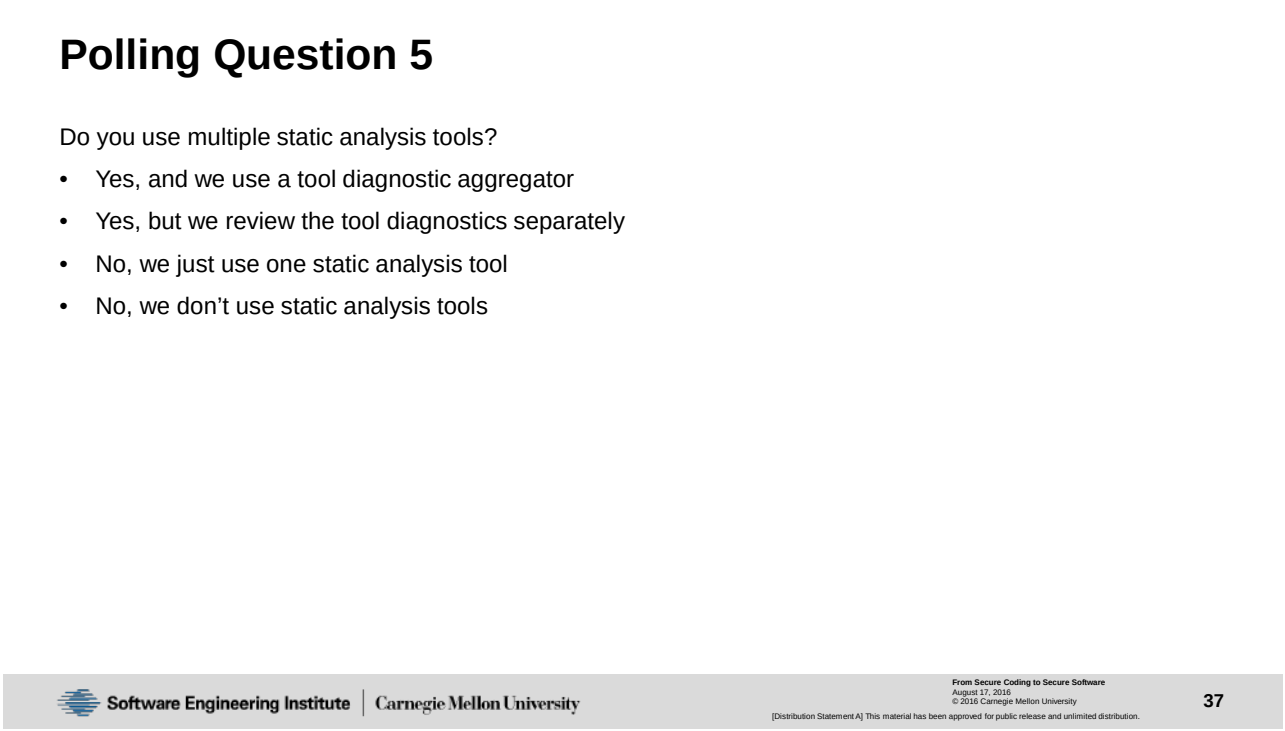
Single view into code and all diagnostics

Maintain record of decisions

**036 Now, of course, if you're running multiple static analysis tools, you have multiple environments where you're getting lots of diagnostics, and so we also recommend using a diagnostic aggregator of some sort. There's a few on the market, and we also have

a research tool and one that we use whenever we do audits ourselves called SCALe, the Source Code Analysis Laboratory, and what that does for us is it-- as you can see from the graphic there-- we run multiple analysis tools and then we run it through SCALe, which first runs it through some secure coding filters to help us prioritize the diagnostics it found related to security as opposed to style or other issues that are unlikely to be affecting security. And then it aggregates all that data so that we can-- all the diagnostics of the tools-- so that we can review and audit the code in one interface and with one data set and database, and it also connects us directly to the source code to easily find the lines of code that the diagnostics are pointing to.

Polling Question 5



**037 And so with that, as I mentioned, I'm curious to find out about our audience, what they're doing currently with static analysis, if they're running multiple static analysis tools and how they are.

Presenter: So that question is posed, and I'll give the results from the last one. The last question was: What testing does your organization perform on your software? We had 44 percent with static analysis, 4 percent with dynamic analysis, 41 percent with both, and 12 percent with none. Okay, so the next one's launched, so back to you.

Presenter: Yeah, so I'll just mention that dynamic analysis-- I'm not going to talk about it too much-- we do have a group here at CERT

that focuses on that. Largely it's the vulnerability team. A lot of that work is in fuzzing, and it's extremely effective at finding real vulnerabilities because you're testing it dynamically after the code is available and you're finding ways that it's vulnerable through paths the software is actually taking, as opposed to looking at the source code that's probably really complex, and seeing defects in the code that may be part of a path that isn't going to be taken. So it's really effective at finding vulnerabilities in code, so we definitely recommend using that as well.

Presenter: And just to close out this question, Bob, we had 52 percent-- the question was: Do you use multiple static analysis tools? Fifty-two percent "No, we just use one tool"; 22 percent, "We just don't use static analysis tools"; 22 percent was "Yes, but we review the diagnostics separately"; and 4 percent, "Yes, and we use a tool diagnostic aggregator".

Presenter: So, I'm sorry, what were the first two.

Presenter: Fifty-two percent at, "No, we use just one static tool."

Presenter: Just one. Okay, so more than half is using at least one, so that's really good. As I said, you should-- and several of the commenters on the chat have said static analysis is a really effective way to find defects.

Select SCALe Assessments

Codebase	Date	Customer	Lang	ksLOC	Rules	Diags	True	Suspect	Diag /KsLOC
A	6/12	Gov1	C++	38.8	12	1,071	52	1,019	27.6
B	3/13	Gov1	C	87.4	28	17,543	86	17,457	200.7
C	10/13	Gov2	C	9,585	18	289	159	130	0.03
D	6/12	Gov3	Java	4.27	18	345	117	228	80.8
E	9/12	Gov2	Java	61.2	33	538	288	250	8.8
F	11/13	Gov2	Java	17.6	21	414	341	73	23.5
G	2/14	Gov4	Java	653	29	8,526	64	8,462	13.1
H	3/14	Gov5	Java	1.51	8	53	53	0	35.1
I	5/14	Mil1	Java	403	27	3114	723	2,391	7.7
J	1/11	Gov3	Perl	93.6	36	6,925	357	6,568	74.0
K	5/14	Gov3	Perl	10.2	10	133	84	49	13.0

**038 And here I just want-- speaking of that specifically. So here's some data of the audits that we have done across several projects-- anonymized, of course-- but it shows that, on average, there's a large disparity across projects of the quality of the code, but on average we've seen about 20 to 30 diagnostics that were true issues per KLOC, or yes, there is significant-- we remove whitespace and some other --and comments and things like that. But the takeaway from this is that just about all code will benefit from static analysis. Regardless of the process that you went through, if you have not done static analysis, you should, because you will find defects.

Polling Question 6

Have you taken some training on secure coding practices?

- Yes, self-taught
- Yes, through an online-delivered program
- Yes, through an in-person delivered program
- Yes, through my academic education
- No

**039 And so here I'm-- so we've talked a little bit about the processes and adopting secure coding standards and tools. I wanted to talk a little bit about training and development of the staff and the developers, but before I do that, I was hoping to see about this polling question about what training there is in common.

Presenter: Yeah, so that question has launched, so maybe we work on one question from Juan while this has launched, Bob, and maybe you covered or not, but I'll ask it. This came in earlier. "How effective is static code scanning-- i.e., Fortify-- to detect bad security practices? How effective is static code scanning?

Presenter: Sure, sure. So that's

the static analysis that I've said. So I've not given quantitative metrics other than showing that it is very effective in finding issues. Again, if you've not done static analysis, you almost definitely find issues by doing it. HP Fortify is one of the top products out there, and what I didn't make clear about SCALe-- I think I might have implied it but didn't make it clear-- SCALe itself is not a static analysis tool; it is only an aggregator. So those boxes in the diagram about static analysis tools, we are using a lot of the same tools that are available to the public, like Fortify and some others, and then we're reviewing those diagnostics. So it is very effective.

Presenter: Okay, we'll just wrap up this polling question here and one other question. The question, real quick, was multiple people asking if an archival recording is available from the talk. An archive of the whole seminar will be available by tomorrow-- same registration URL that you used today that you can watch the archive. So the question was: Have you taken some training on secure coding? Thirty-one percent yes, self-taught; 12 percent, yes, through an online-delivered program; 14 percent yes, through an in-person delivery program; 5 percent yes, through an academic education; and 38 percent no.

Presenter: Okay, great, and thanks everyone for being honest. So it's actually a little better than I thought. I like to say, generically, in today's

world, because it's so easy to program that a lot, if not a majority, of developers, people that are software developers, have not been properly trained in software development, and almost none of them have gone through secure coding training. So it's actually-- even the self-taught people I think are better off. It shows that you have an interest in learning about secure coding practices, and it's a lot more effective than not having any. So for the-- I think it was 38 percent-- the 38 percent that was no, I would definitely recommend at least trying something online as a MOOC or something else, and if you have a large group, we have instructor-led training. We also, if you want something more formal-- and I'll talk about this in a minute-- but we have online training as well.

Secure Coding Professional Certificates

**040 And so to kind of roll into that, as I was mentioning there, we do have online training that ends with a secure coding professional certificate, and this training is in the C, C++-- so one edition of the training is C and C++ combined, and the other is Java, and each of those certificate programs have two courses, a secure software concepts course and then a secure coding in the particular language course, and then it ends with a completion exam. And then we also, as I mentioned, for larger groups at an organization, we have instructor-led courses onsite as well.

SEI Secure Coding in C/C++ Training 1

The Secure Coding course is designed for C and C++ developers. It encourages programmers to adopt security best practices and develop a security mindset that can help protect software from tomorrow's attacks, not just today's.

Topics

- String management
- Dynamic memory management
- Integer security
- Formatted output
- File I/O

http://www.sei.cmu.edu/training/p63.cfm

**041 Here I'll talk just for a minute about some of the topics and the objectives of the course, and these are really topics and objectives that you'd look for any course that you were going to take, and training. These are kind of the common issues. So secure coding in C and C++ obviously really important for string management-- string meaning array, meaning buffers and buffer overflows. Dynamic memory management is another big issue, and pointers and properly freeing memory. You have integer security, as Mark was mentioning-- wrapping of integers and that, especially if it's in pointer arithmetic, that being an issue, and formatted to output, that being format strings, and then file I/O.

SEI Secure Coding in C/C++ Training 2

Participants gain a working knowledge of common programming errors that lead to software vulnerabilities, how these errors can be exploited, and mitigation strategies to prevent their introduction.

Objectives

- Improve the overall security of any C or C++ application.
- Thwart buffer overflows and stack-smashing attacks that exploit insecure string manipulation logic.
- Avoid vulnerabilities and security flaws resulting from incorrect use of dynamic memory management functions.
- Eliminate integer-related problems: integer overflows, sign errors, and truncation errors.
- Correctly use formatted output functions without introducing format-string vulnerabilities.
- Avoid I/O vulnerabilities, including race conditions.

**042 And the objectives are to improve the overall security of the C and C++ applications that you're developing; avoiding vulnerabilities by learning what, again, the constructs in C and C++ are that you shouldn't use, or how to use them appropriately, because often just calling a particular method or function is not the right way to do it, that there's other checks that you need to put before or after you call functions to protect the software, and this teaches you how to do that.

Java Secure Coding Course

The Java Secure Coding Course is designed to improve the secure use of Java. Designed primarily for Java SE 8 developers, the course is useful to developers using older versions of the platform as well as Java EE and ME developers. Tailored to meet the needs of a development team, the course can cover security aspects of

Trust and Security Policies	Object Orientation	Serialization
Validation and Sanitization	Methods	The Runtime Environment
The Java Security Model	Vulnerability Analysis Exercise	Introduction to Concurrency
Declarations	Numerical Types in Java	in Java
Expressions	Exceptional Behavior	Advanced Concurrency
	Input/Output	Issues

http://www.sei.cmu.edu/training/p118.cfm

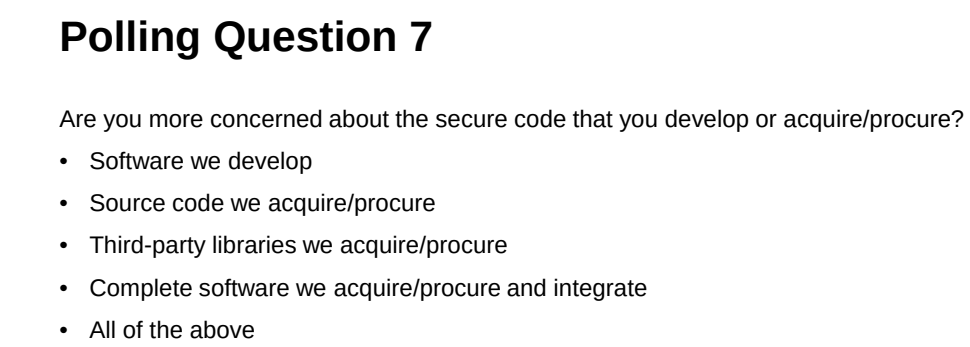
**043 And then the Java Secure Coding course, as I mentioned. So a lot of people immediately think, "Well, Java is secure as is. It has memory protection and it has a lot of other protections." Well, as it turns out, there's a lot of different ways that you can misuse Java. Not the same ways as C and C++, but you can get into trouble. I'll just mention the one in the top right there. Serialization and deserialization has been a very-- recently come out as a trouble spot with Java and using it correctly. So there definitely are issues there. Go ahead, Mark.

Presenter: And just to illustrate the comments that people were making before with SQL injection, even within Java you have the same issue that is a vulnerability whenever you

open up the capability to execute arbitrary code. In the case of Java, they call it class loaders, but nevertheless it's the same paradigm that we see repeated in languages again and again, and how you protect against it has the same generic answer; the details depend on the language.

Presenter: Yeah, that's right, interfacing-- using Java to interface with other languages and using the constructs of Java correctly are not always obvious, and so this teaches you how to use those APIs correctly and in a secure way.

Polling Question 7

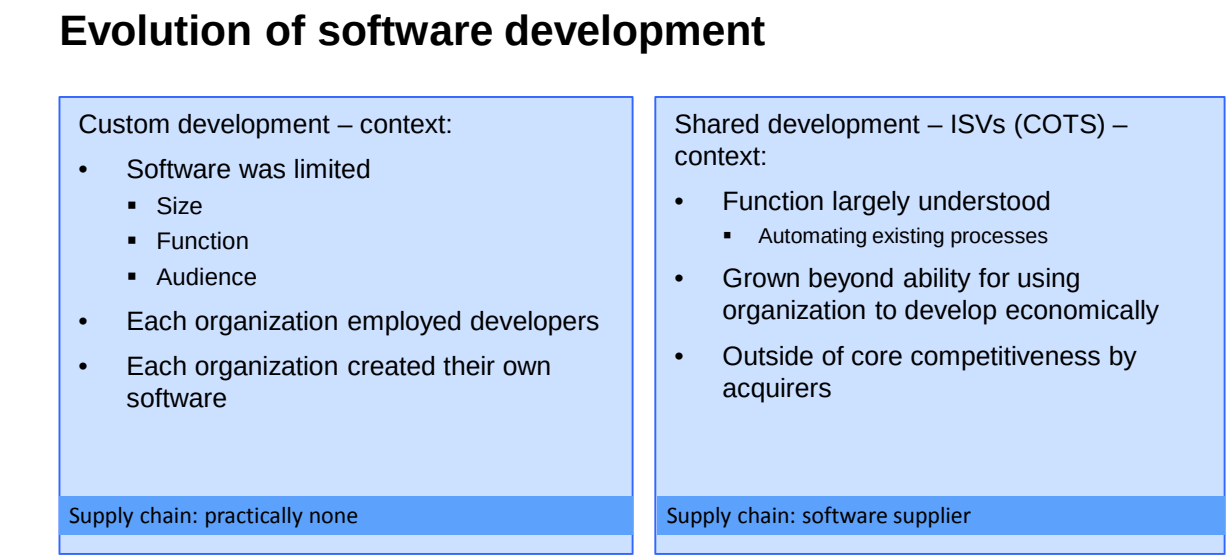


**044 And so that covers most of what I'm going to talk about of developing software. Of course that

doesn't address everybody's concern-- well, nothing does-- but there's a big area that it doesn't address, and that is that a lot of people acquire software or acquire source code or libraries that have been compiled, and they're using third-party software but they're not sure what's in it. And so first I'd like to find out from this polling question what people's proportion of building versus buying, so to speak, even though I know that often it's open source and free.

Presenter: So we've got 34 percent software we develop; 3 percent source code we acquire or procure; 10 percent third-party libraries; 1 percent complete software we acquire; 52 percent all of the above.

Evolution of software development



**045 Presenter: Okay. So a lot of integrating pieces and packages from a lot of different places. So I was wondering, Mark, could you talk a little bit about acquiring software securely?

Presenter: Sure. And also we'd like to just react to a couple of the questions that came by. First of all, there was a fair amount of discussion about SCALe, about the aggregator that we talked about, and clearly we have been asked to include information about where to find that in reports and so on. So we'll include that in the website rather than trying to give all those URLs on the fly here.

Presenter: Right. Right.

Presenter: There was also a discussion about compiler optimizations, and someone used the word "undesired". Our technical team actually is really fascinated by that particular topic, and they've put together a whole other presentation and seminar, which, if you want, by all means, let Shane know, but we call it "unexpected compiler optimizations" because all those optimizations were put there for a reason, and certainly in many circumstances they were desired. The question is whether you expect them or not, and the broad-brush comment we'd make is either-- one set of problems happens when people really don't understand the language in enough precision. The other circumstance where it runs into problems is in portability, in where they do understand the language and

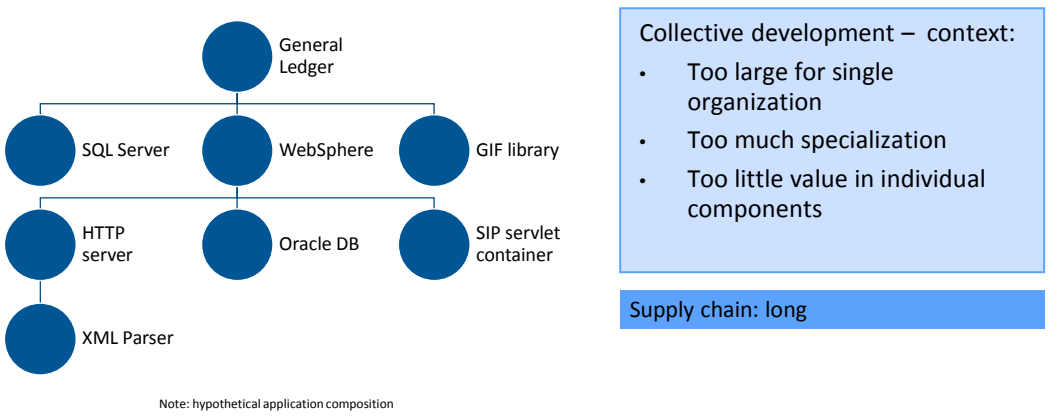
the correct precision, but they move it from one system to another system, and a common one that we've seen is moving from 16-bit to 32-bit machines, and all of the sudden optimizations which they thought-- assumptions and implementations they thought-- no longer hold and they run into problems. But that's a whole other talk. There's a whole lot more to be said. We'll defer that to another time, if you'd like to hear about it.

But one of the topics that we've run into when we talk with development organizations about the problems that they face, part of it is-- what we've discussed here-- is the code that they are writing, and historically that really was the focus of attention. So when software started being developed-- and I'll use my own sister as an example. She worked for a large manufacturing company and one of the jobs she had to do was to build an airline reservation system. No, she didn't work for Delta or United or whatever. She worked for a company that had plants and extrusion lines and so on, but they had a bunch of corporate jets and they needed to schedule them, and so she went and built an airline reservation system from scratch.

In that kind of context, it was custom development, limited amount of software, limited audience, limited function, and for those purposes, companies like that one basically employed their own developers and they created their own software.

They developed everything. That turned out to be fairly expensive, and quickly they decided that there were some things which really weren't their core differentiation, and so the industry of independent software vendors came up, and they were automating well-known processes. I don't know about airline reservations being so ubiquitous, but things like general ledger work, enterprise resource management, supply chain management-- the whole variety of common kinds of functions that needed to be done that were not the specialty of any one company, and so you had Infor Global, SAP, Oracle-- a whole variety of companies who went and built this, and now you started getting a supply chain-- a very small supply chain-- it was just the vendor that you dealt with.

Development is now assembly



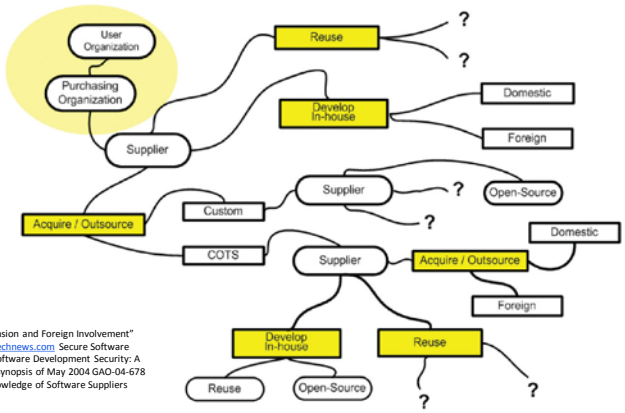
**046 What has evolved over time is that actually there's very little development being done on large programs outside of these independent software vendors. I mean, obviously Oracle or Microsoft spend a great deal of their time building programs completely from scratch. But for most places, development is assembly. It's simply too large for any individual organizations. There's too much specialization in each individual component, and frankly, every little component doesn't have enough value to invest in it, whether it's a SIP container to connect to telephony systems, your own database, your own XML parser, and so on.

Software supply c hain for assembled software

Software supply chain for assembled software

Expanding the scope and complexity of acquisition and deployment

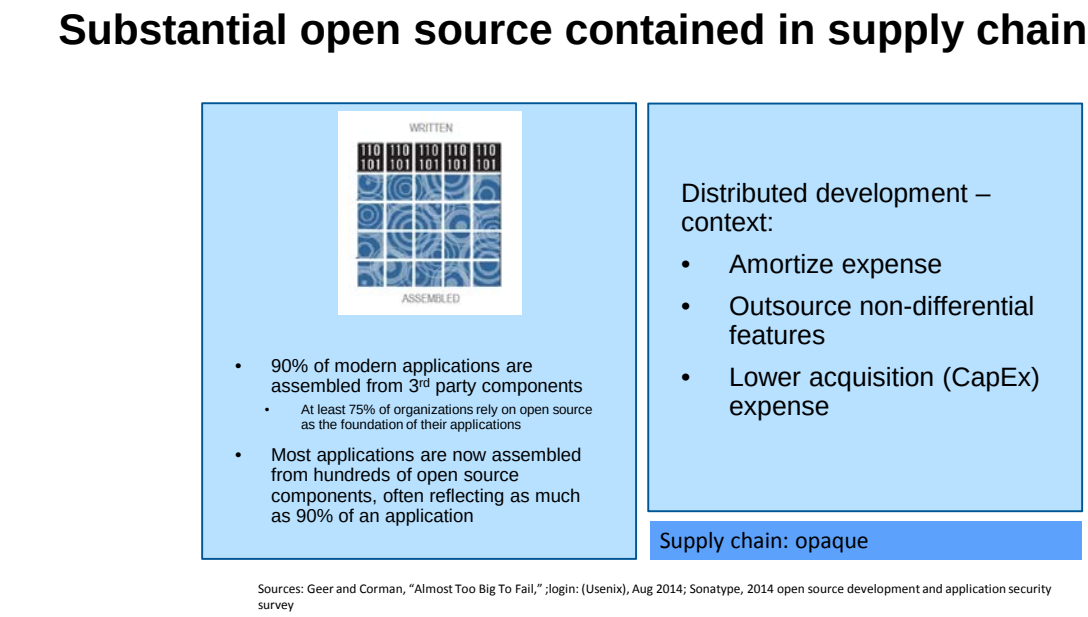
Visibility and direct controls are limited (only in shaded area)



Source: "Scope of Supplier Expansion and Foreign Involvement" graphic in DACS www.softwaretechnews.com Secure Software Engineering, July 2005 article "Software Development Security: A Risk Management Perspective" synopsis of May 2004 GAO-04-678 report "Defense Acquisition: Knowledge of Software Suppliers Needed to Manage Risks"

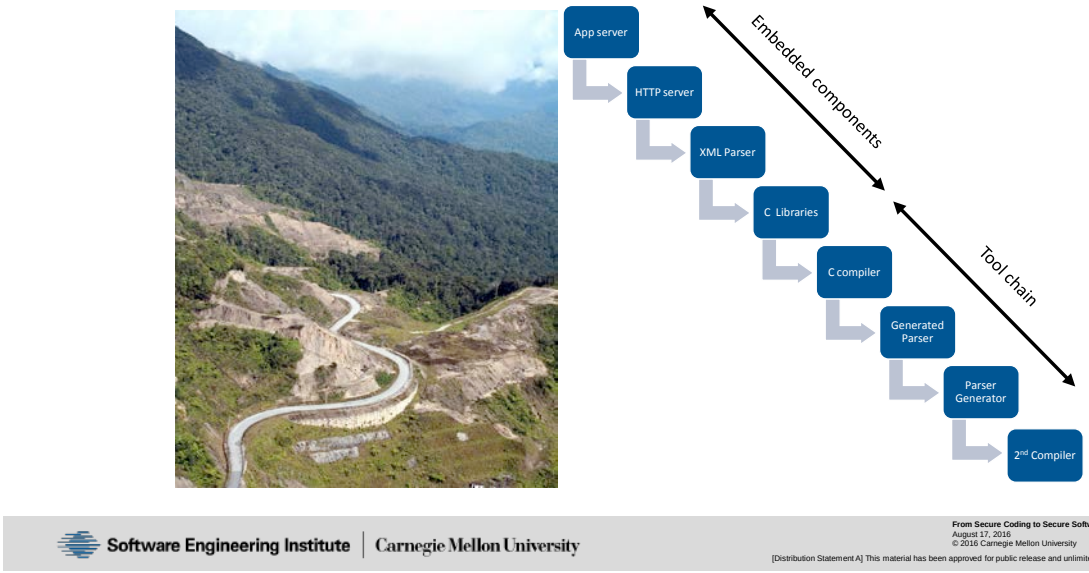
**047 So instead, we've wound up as having a large supply chain, in where you reach out to lots of vendors, and they reach out to vendors, and they reach out to vendors.

Substantial open source contained in supply chain



**048 But in this supply chain, you have a large amount of open source. So what's the magnitude of this? Well, when we've done-- I shouldn't say we've done-- we rely on third-party surveys-- they found that about 90 percent of applications, in fact, are assembled. They're not constructed. And perhaps as importantly, of those assembled applications, 90 percent of their content comes from the outside, and so now you have a very long supply chain of people getting pieces from other people. It's very long.

Open source s upply chain has a long path



**049 Just to give you an example, let's say you have an application server. That contains an HTTP server that came from another place, which has an XML parser which came from another place, which came with some C libraries that came from another place that was generated by a compiler that came from another place. The compiler itself was generated by a parser generator that came from another place, and so on. You wind up that there's a huge long list of dependencies.

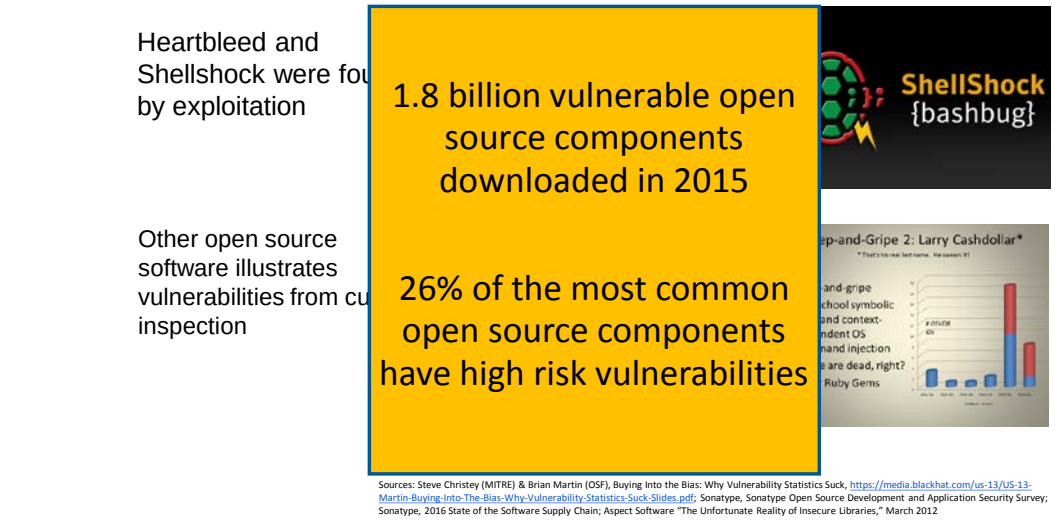
Corruption in the tool chain already exists

Sources: http://www.macrumors.com/2015/09/24/xcodeghost-top-25-apps-apple-list/
http://www.itntoday.com/2015/09/the-85-ios-apps-affected-by-xcodeghost.html

- XcodeGhost corrupted Apple's development environment
- Major programs affected
  - WeChat
  - Badu Music
  - Angry Birds 2
  - Heroes of Order & Chaos
  - iOBD2

**050 And there are problems. You might think that, "Who's going to really screw around with a compiler?" Well, it's happened already. For example, Apple had their development environment attacked and, as a result, they had applications built with Xcode being corrupted.

Open source is not secure

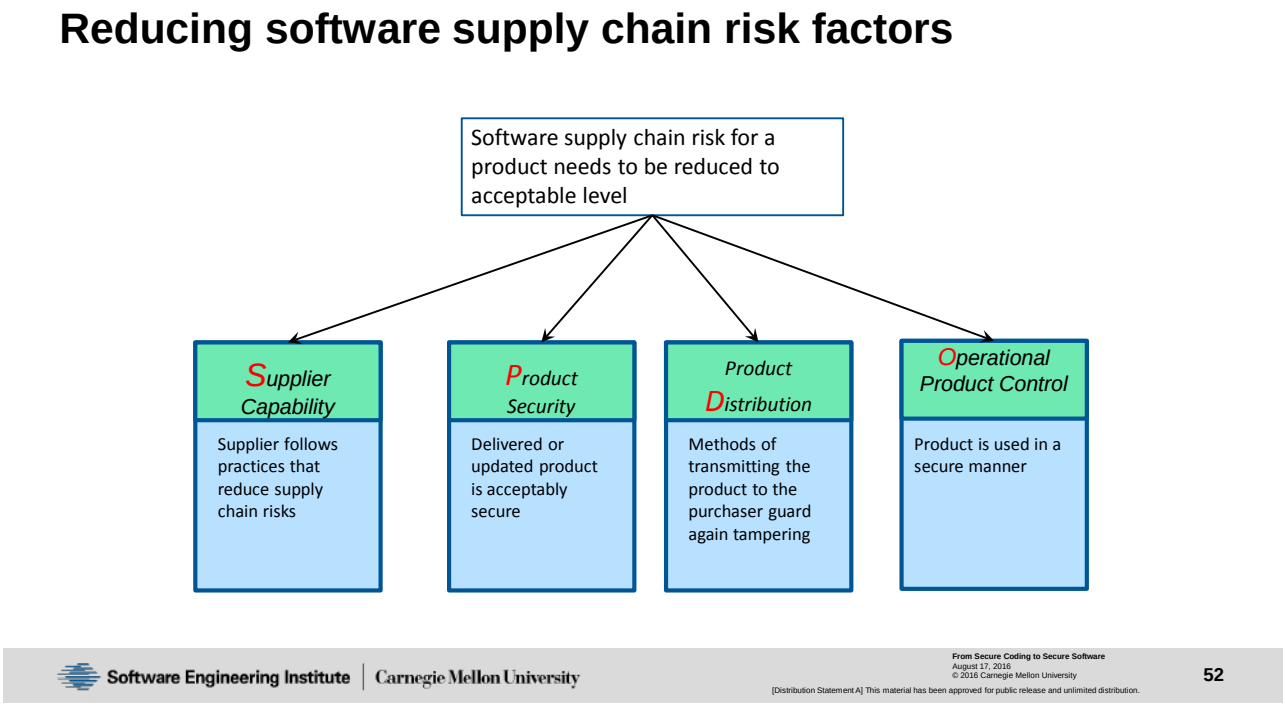


**051 And open source is not secure. We use a lot of it. People know about Shellshock and bashbug, just as two common examples that happened because people were explicitly exploiting them. But worse, a study that was presented at Black Hat showed that finding bugs was basically shooting fish in a barrel, and they were quoting some researchers here which said, "We just decided to go look for a particular kind of bug," and they found so many that it was upsetting all the statistics that were being used.

Now, to quantify this, rather than just saying, "Well, there's Shellshock," and whatever, again, there have been studies done just in 2015, close to 2 billion open-source components with serious vulnerabilities were

downloaded. Twenty-six percent. A quarter of the most open-source components have high-risk vulnerabilities. If you're using Spring, you probably are one of these people.

Reducing software supply chain risk factors



**052 How to do that. How to reduce this. Well, by managing your software supply chain-- and the methodology we use is called SPDO, very minimal methodology, if you'd like to remember it that way-- making sure your supplier knows how to do secure coding, that their product was built correctly, that it wasn't modified in distribution, and that it's in the right operational context.

Now, for each of these, we have

specific recommendations that you can follow.

Supplier security commitment evidence

Supplier security commitment evidence

Supplier employees are educated as to security engineering practices

- Documentation for each engineer of training and when trained/retrained
- Revision dates for training materials
- Lists of acceptable credentials for instructors
- Names of instructors and their credentials

Supplier follows suitable security design practices

- Documented design guidelines
- Has analyzed attack patterns appropriate to the design such as those that are included in Common Attack Pattern Enumeration and Classification (CAPEC)
- Application of code signing techniques (interest in ISO 17960 – in early draft)

**053 For supplier evidence, do they really know what they're doing? And again, you can look at the details on replay.

Evaluate a product's threat resistance

What product characteristics minimize opportunities to enter and change the product's security characteristics?

- Attack surface evaluation: Exploitable features have been identified and eliminated where possible
  - Access controls
  - Input/output channels
  - Attack enabling applications – email, Web
- Design and coding weaknesses associated with exploitable features have been identified and mitigated (CWE)
- Independent validation and verification of threat resistance
- Dynamic, Static, Interactive Application Security Testing (DAST, SAST, IAST)
- Delivery in or compatibility with Runtime Application Self Protection (RASP) containers

**054 Similarly, how to evaluate a product that you're getting, whether it's been properly put together or not.

Establishing good product distribution practices

Recognize that supply chain risks are accumulated

- Subcontractor/COTS-product supply chain risk is inherited by those that use that software, tool, system, etc.

Apply to the acquiring organizations and their suppliers

- Require good security practices by their suppliers
- Assess the security of delivered products
- Address the additional risks associated with using the product in their context

Ideally open source is built with a compiler you trust

**055 How do you know that it has been distributed in a way that hasn't changed along the way?

Maintain operational attack resistance

Who assumes responsibility for preserving product attack resistance with product deployment?

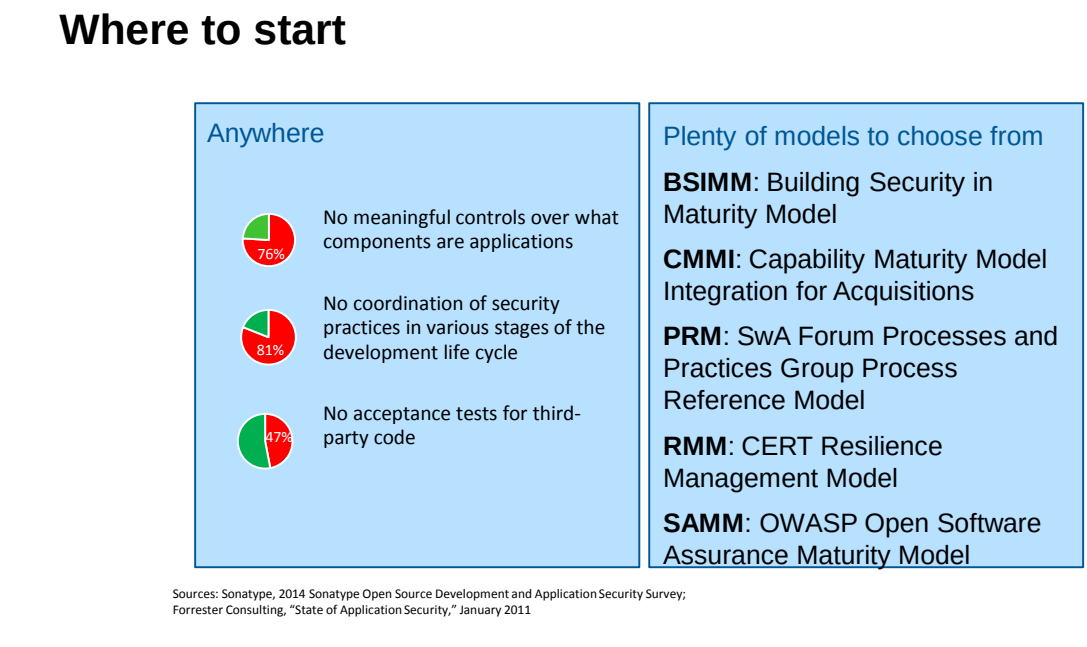
- Maintaining inventory of components
- Patching and version upgrades (component lifecycle management)
- Expanded distribution of usage
- Expanded integration

Usage changes the attack surface and potential attacks for the product

- Change in feature usage or risks
- Are supplier risk mitigations adequate for desired usage?
- Effects of vendor upgrades/patches and local configuration changes
- Effects of integration into operations (system of systems)

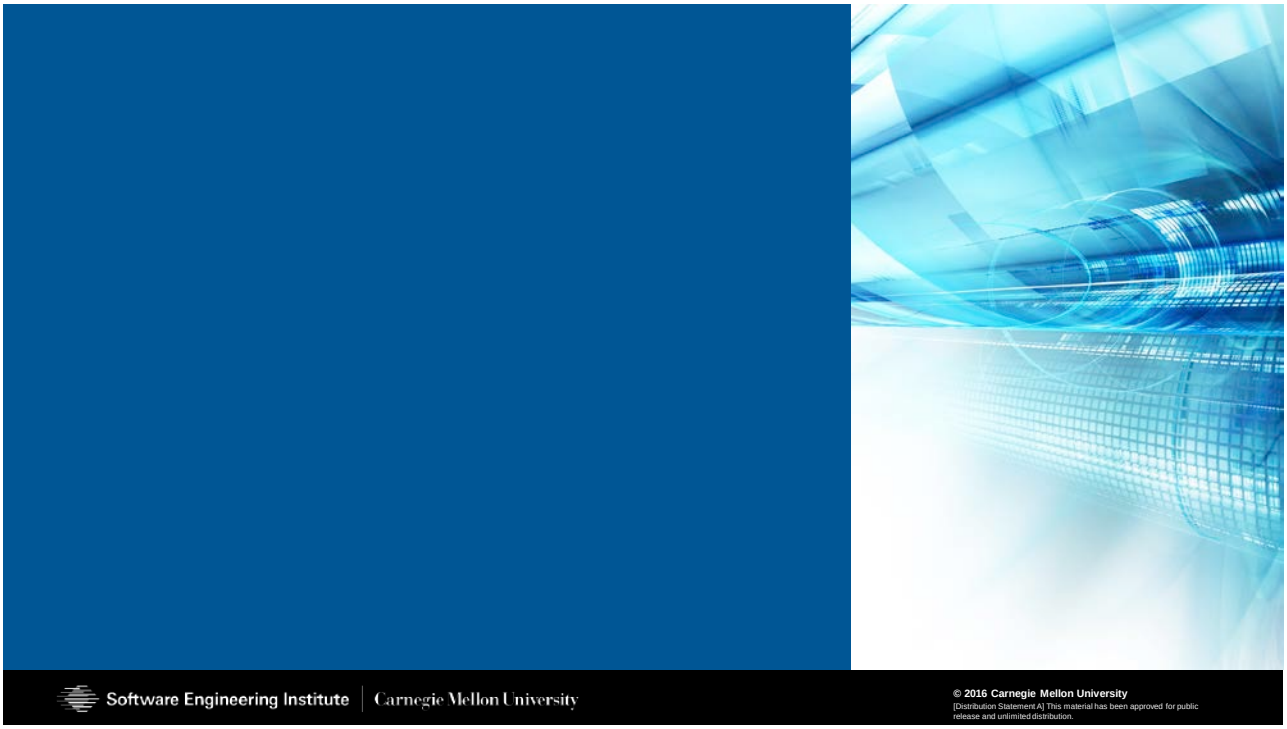
**056 And perhaps most importantly, that you haven't changed the operational environment. Most of the problems that we see in large systems have the consequence that they were built under one set of security assumptions, put into a different environment, and they've changed the security assumptions.

Where to start



**057 So, we've said an awful lot of things you can do, and you might say, "Well, what should I go after first?" The unfortunate truth is that there is so much to do that you can pretty much start anywhere. On the left you can see the large fractions of people who basically have huge security gaps in their development processes, and there are lots of ways to do this, and on the right we list a variety of choices that you can use depending on if you like us, you can use things like RMM, comes from CERT. If you like what Cigital does, you can use BSIMM. But there are many alternatives-- each one has their pluses or minuses, but there's such a gap that pick one and use it as you would like.

Questions



**058 With that, I'm going to let Bob close here.

Presenter: Sure. Thanks Mark, and sorry for-- well, thank you for doing a good job of kind of getting through that. Sorry to the audience for him having to do that. If you have questions, please let us know, and we'll be sure to follow up. With that, that pretty much closes, except for questions, and there are two things I wanted to say really quickly. One was that-- where are we going with research. Just quickly, we're looking at ways to use machine learning to improve the accuracy of static analysis tools. In particular, can we use the data of what we've found with diagnostics that are either true-positives or false-positives, and use other information to improve our

prediction of whether or not it's a true-positive or false-positive and something to pay attention to; as well as working on automatically correcting code, finding code that an analyzer might find as a diagnostic but that we feel-- we're pretty sure that it is a defect and informing the developer on the spot what could be done to correct the defect, or for code that is out in the wire, just fixing the code with near-perfect or perfect accuracy to not cause any problems.

The last thing I wanted to mention was that we do have a secure coding symposium coming up if you're looking for more information about secure coding, both from the SEI and from the community. On September 8 we'll be in Arlington, Virginia, at the Secure Coding Symposium. Largely the day-- it's a one-day event. It will be a few keynotes and several panel discussions and then a tutorial. The keynotes will be outside speakers; the panels will be a mix of SEI and outside speakers from government and industry. And we'll end with a tutorial. And the registration page is already available, and I believe it's in your resources areas. There's a general agenda listed as well, and we're going to be updating that within the next couple days with the names of speakers and the presenters.

Presenter: And I'll just add there's no cost to attend that but space is limited, and a lot of space is already taken up.

Presenter: That is correct. That is correct.

Presenter: So if you have interest, make sure you click on the link in the Resource tab. So we know it is two thirty, so if you have to go, we understand, but I'd like to just get one question before we wrap up, and that's from Brian, asking, "Would you agree that turning up compiler warnings to a high level, e.g., W4, should be a priority to secure a legacy code base?"

Presenter: It depends who you ask. This goes back to the false-positive comment, that it will definitely increase the number of diagnostics that get generated, and the question becomes at what point do your developers start ignoring all the diagnostics. We've gotten stories from development organizations like Google who tell us that if they put out any false diagnostics the developers revolt and they can't put them out at all. Other places are far more rigid and say you've got to address every single diagnostic that comes out of every single tool, and they pay the price for it, but they feel they get the value, and that's an organizational choice.

Presenter: Yeah, yeah. Agreed, agreed.

Presenter: Okay, just a couple closing comments from me. So we had a number of comments through the chat-- so great participation there-- and a number of questions

we didn't get to. There is a secure coding forum on LinkedIn. We ask that everybody join that forum through LinkedIn. Just search for the secure coding forum within the groups and you can join the group and continue the conversation there.

As Bob mentioned, the symposium is coming up. It's in your Download Materials tab, that you can get information on that. Of course we ask you to fill out the survey upon exiting today's event, as your feedback is always greatly appreciated.

And lastly, the next webinar we'll have is going to be on September 14. The topic will be building and scaling a malware analysis system by Brent Fry. So that's all we have. Thanks again for everyone's participation today. Have a great day.