UNDERSTANDING SECURITY MISTAKES DEVELOPERS MAKE

Qualitative Analysis From Build It, Break It, Fix It

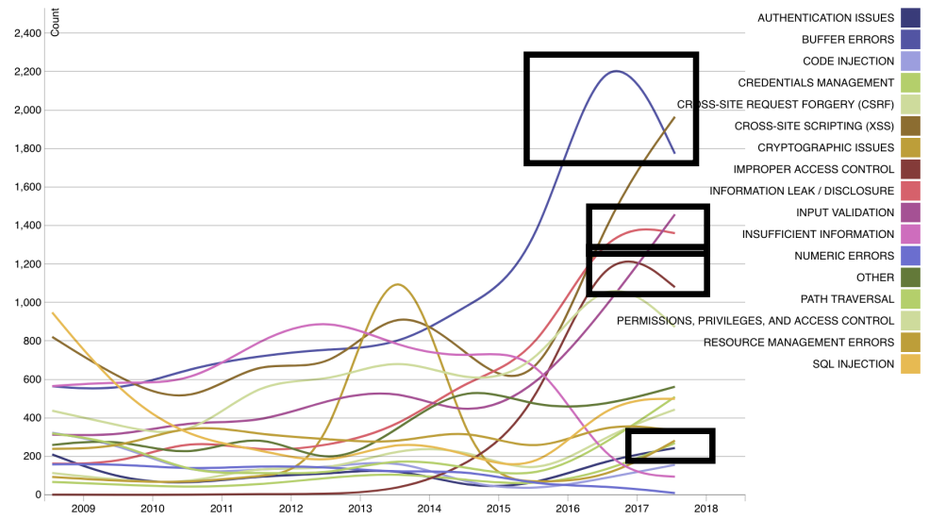
Daniel Votipka, Kelsey Fulton, James Parker, Matthew Hou, Michelle Mazurek, and Mike Hicks

University of Maryland, College Park

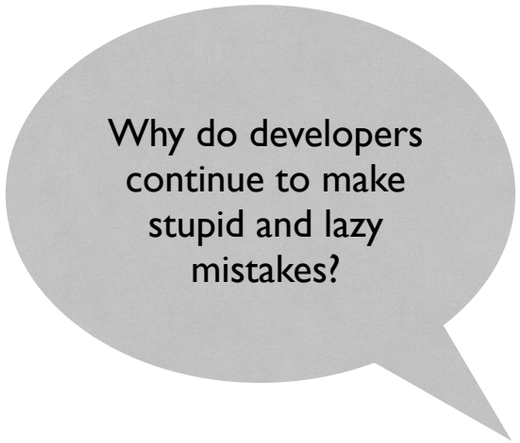
“SOLVED”VULNERABILITIES ARE STILL A VERY REAL PROBLEM

Vulnerability Type Change By Year

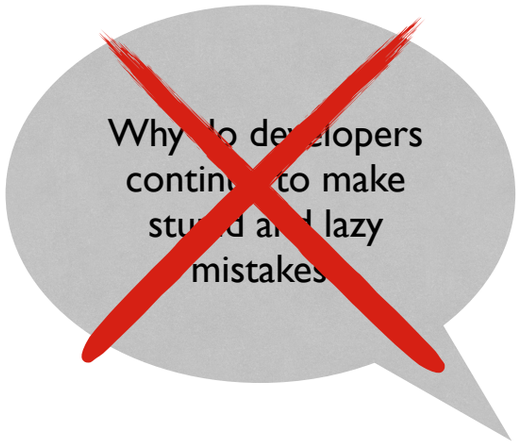
This visualization is a slightly different view that emphasizes how the assignment of CWEs has changed from year to year.



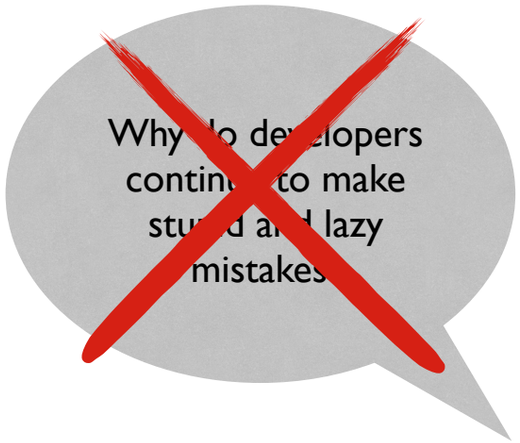
Here we’re looking at the vulnerabilities type change by year based on cwe reporting from NIST. Note that buffer errors still remains in a top 2 spot. information leak/ disclosure, improper access control, and cryptography issues have barely or not at all decreased, and yet they’re supposed to be relatively “solved” issues given current libraries and APIs. *Note I’m using the word solved here very loosely. So if we have solutions to this problem, why are we still seeing these vulnerabilities in the real world? We can see some of these play out in recent examples.



So some experts may ask the question:



There’s a flaw in this thinking though. We cannot expect developers to know how to do everything, and we cannot assume the all the fault can be placed on them. But we need to avoid thinking this way. Of the developers as the enemy. Instead we should ask



How can we make secure programming easier?

POSSIBLE SOLUTIONS

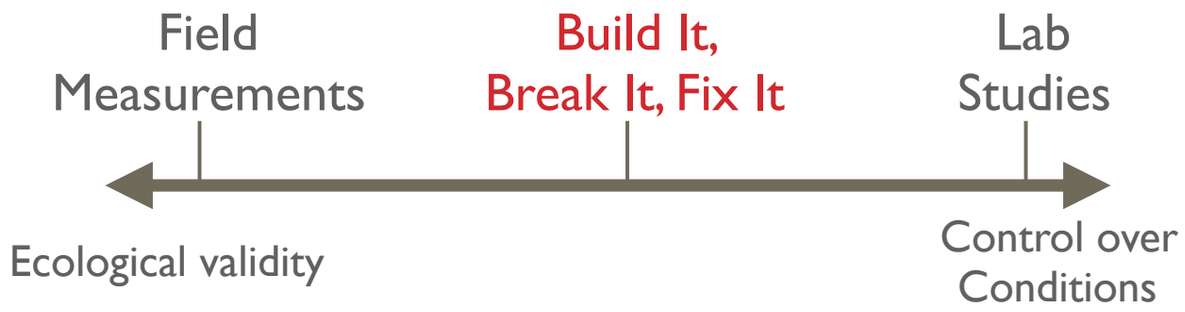
- More/Better Education
- Better APIs
- Better documentation
- Automation
- Etc.

How can we improve the effectiveness of these solutions?

So we have some classic existing solutions such as: Obviously, these aren’t working as well as we’d hoped. So we may be asking ourselves how can we improve the effectiveness of these solutions?

IN ORDER TO IMPROVE THESE SOLUTIONS, WE NEED TO UNDERSTAND THE **TYPES, CAUSES, AND PERVASIVENESS** OF VULNERABILITIES.

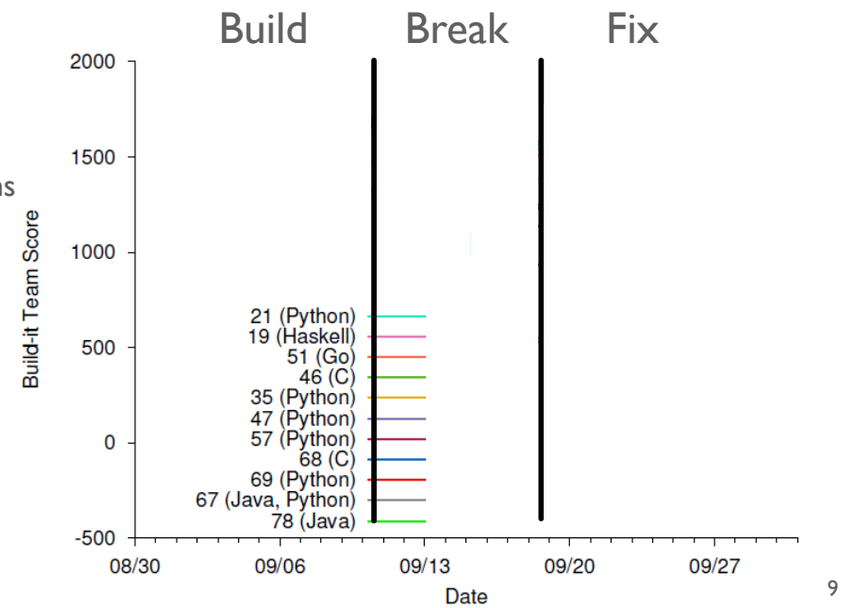
HOW TO MEASURE?



How can we measure this. Well, when deciding a method for this type of study, you're working along a spectrum with ecological validity at one end and control over conditions at the other.

BUILD IT, BREAK IT, FIX IT

- Build to spec
- Then break other teams
- Incentivize:
  - Functionality
  - Performance
  - Security



Ruef et al., CCS 2016

contest. We take all of these approaches and balance the tradeoffs.

Build it – two weeks, build to spec, any language, any design choices
Break it – given source, try to find correctness and security bugs in any other teams' code

RESEARCH QUESTIONS

- What types of vulnerabilities do developers introduce?
- How severe are the vulnerabilities? If exploited, what is the effect on the system?
- How exploitable are the vulnerabilities? What level of insight is required and how much work is necessary?

14:00

RESEARCH QUESTIONS

- What types of vulnerabilities do developers introduce?
- How severe are the vulnerabilities? If exploited, what is the effect on the system?
- How exploitable are the vulnerabilities? What level of insight is required and how much work is necessary?

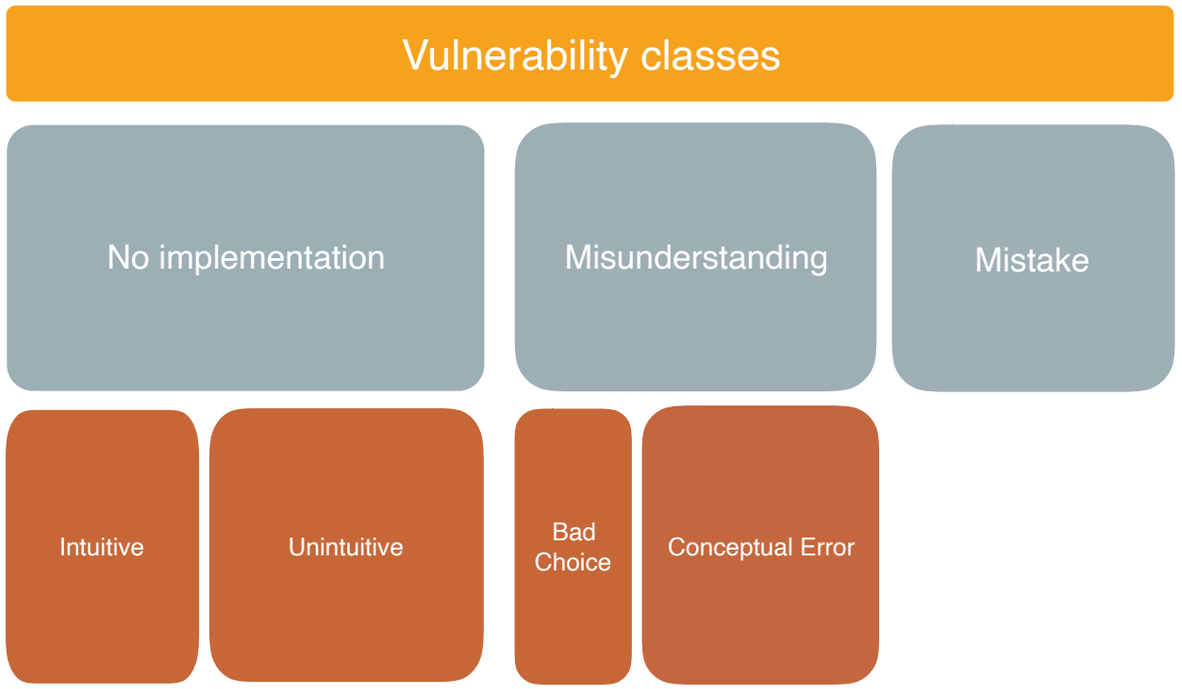
14:00

ANALYSIS APPROACH

- Examine projects and associated exploits in detail
- Iterative open coding
  - Two independent researchers with high reliability
- 94 projects with 866 submitted exploits
- Both qualitative and quantitative analysis performed

Vulnerabilities are both the breaker submitted and researcher identified. We used a rigorous technique that is considered best practice that was developed in the social sciences.

RESULTS



So from our coding process, we developed these classes. I’m going to explain what each of these mean later.

Vulnerability classes

No implementation

The no implementation class can be broken into three subclasses

Vulnerability classes

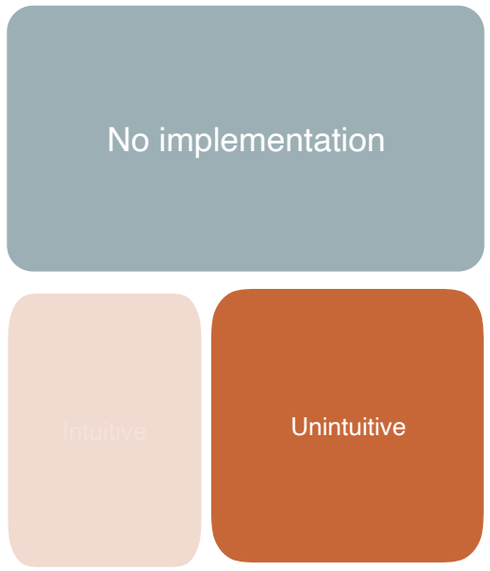
No implementation

Intuitive

- Missed something "Intuitive"
  - No encryption
  - No access control

The first one is all intuitive where a team missed something that would be considered to be completely intuitive and necessary to implement. Examples of this are….

Vulnerability classes



- Missed something "Intuitive"
  - No encryption
  - No access control
- Missed something "Unintuitive"
  - No MAC
  - Side-channel leakage
  - No replay prevention

The last subclass is unintuitive where a team missed implementing something that may not necessarily be obvious or intuitive to implement such as no MAC (integrity checks), checking for side-channel leakage, or making sure to prevent a replay attack

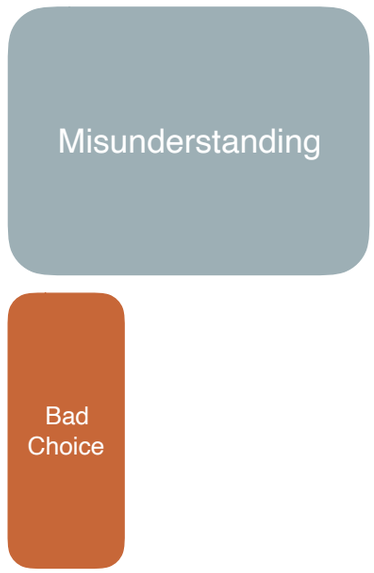
Vulnerability classes

Misunderstanding

The next class I'll talk about is the misunderstanding class. It can be broken into two subclasses.

Vulnerability classes

- Made a "Bad Choice"
  - Weak algorithms
  - Homemade encryption
  - strcpy



The first subclass is defined by teams that made a bad choice in their security implementation and it resulted in a vulnerability. One code in this subclass is choosing weak algorithms.

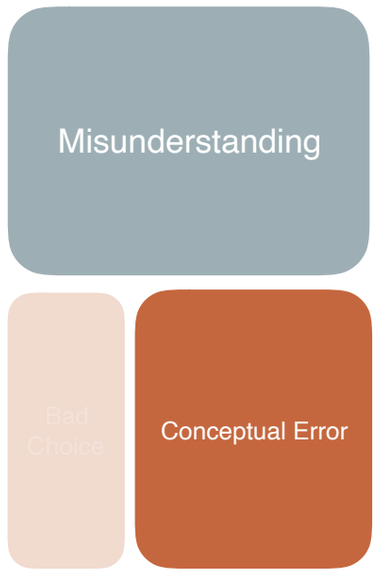
Another is using homemade encryption. An example of a team using homemade encryption is one team xor'd key length chunks of the text with the user provided key to make the final ciphertext. Therefore, an attacker could simply extract two key length chunks and xor them to get the key.

Another category is using functions like strcpy. One team used strcpy and missed a single bounds check. Rather than classifying this as a mistake, we chose bad choice because they could have used the bounded copy in the first place. A final code within this category is having a weak access control design.

Weak access control design example == The default delegator's permissions are checked at use-time, not creation time.

Vulnerability classes

- Made a "Conceptual Error"
  - Fixed value



The second subclass is defined by teams that made an error in the conceptual way they were thinking. One code within this subclass is using a fixed value.

```
1  def fillercrypter(sharedkey, text):
2      ...
3      encryption_suite = AES.new(sharedkey,
4        AES.MODE_CBC, 'This is an IV456')
5      ...
```

From this site I have this code snippet:

8

```
>>> from Crypto.Cipher import AES
>>> obj = AES.new('This is a key123', AES.MODE_CBC, 'This is an IV456')
>>> message = "The answer is no"
>>> ciphertext = obj.encrypt(message)
>>> list(bytearray(ciphertext))
[214, 131, 141, 100, 33, 86, 84, 146, 170, 96, 65, 5, 224, 155, 139, 241]
```

asked 5 years, 2 months ago
viewed 2,188 times
active 5 years, 2 months ago

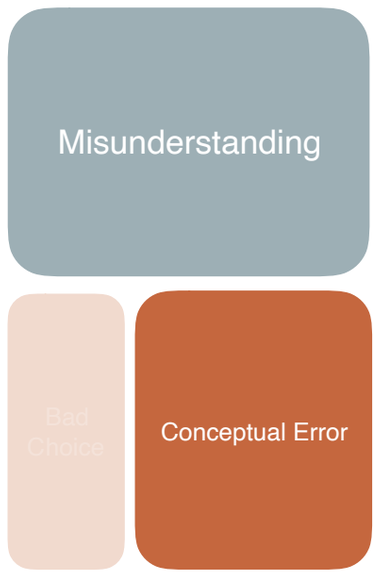
BLOG

An example of this code can be seen with this team using a fixed IV value. Turns out that this is…

Actually a stack overflow answer

Vulnerability classes

- Made a "Conceptual Error"
  - Fixed value
  - Lacking sufficient randomness
  - Disabling protections in library



Other codes within this subclass are lacking sufficient randomness, using security on only a subset of the data instead of all of it, and intentionally disabling protections provided by a library. An example of this last code can be seen in….

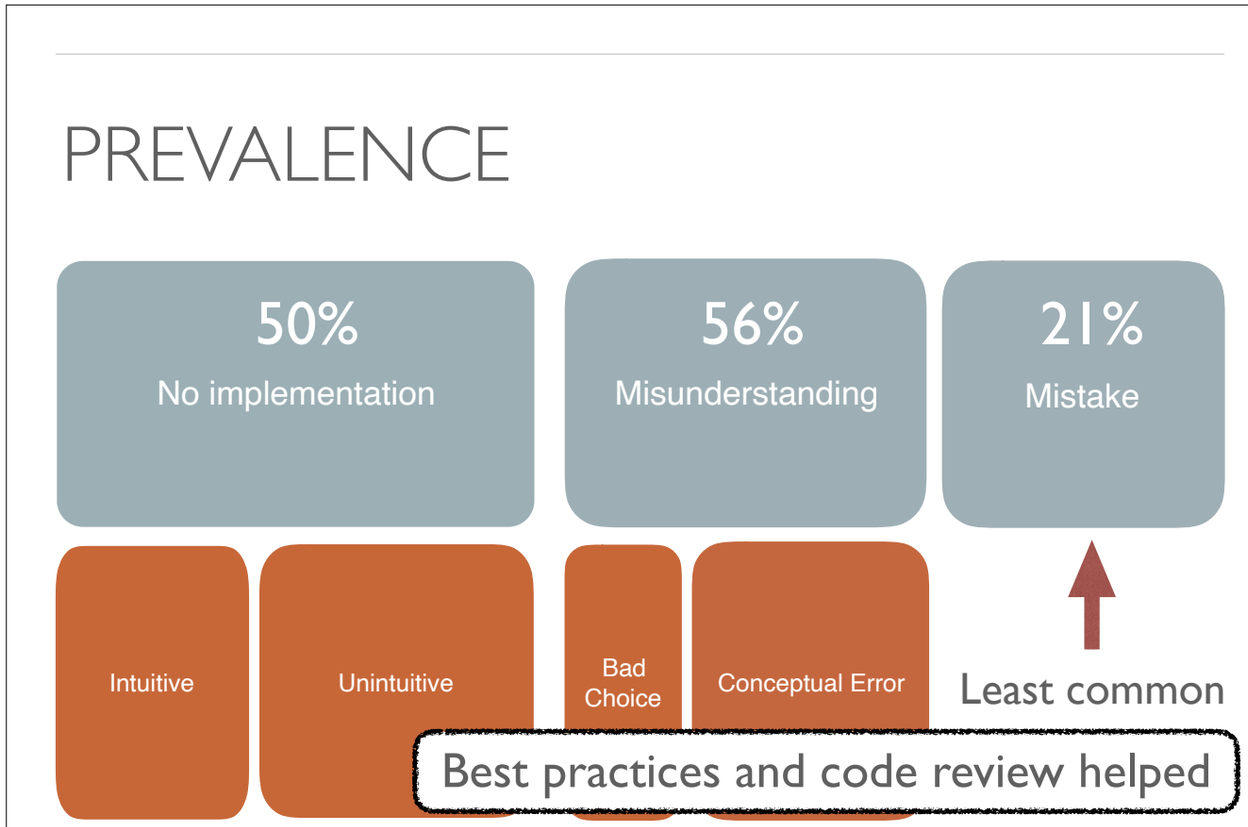
```
1  self.db = self.sql.connect(filename, timeout=30)
2  self.db.execute('pragma key="' + token + '";')
3  self.db.execute('PRAGMA kdf_iter='
4    + str(Utils.KDF_ITER) + ';')
5  self.db.execute('PRAGMA cipher_use_MAC = OFF;')
6  ...
```


This team who used sqlcipher which has full page MACing built in. This team decided to explicitly turn the use of a MAC off within the library for a non-obvious reason.

Vulnerability classes

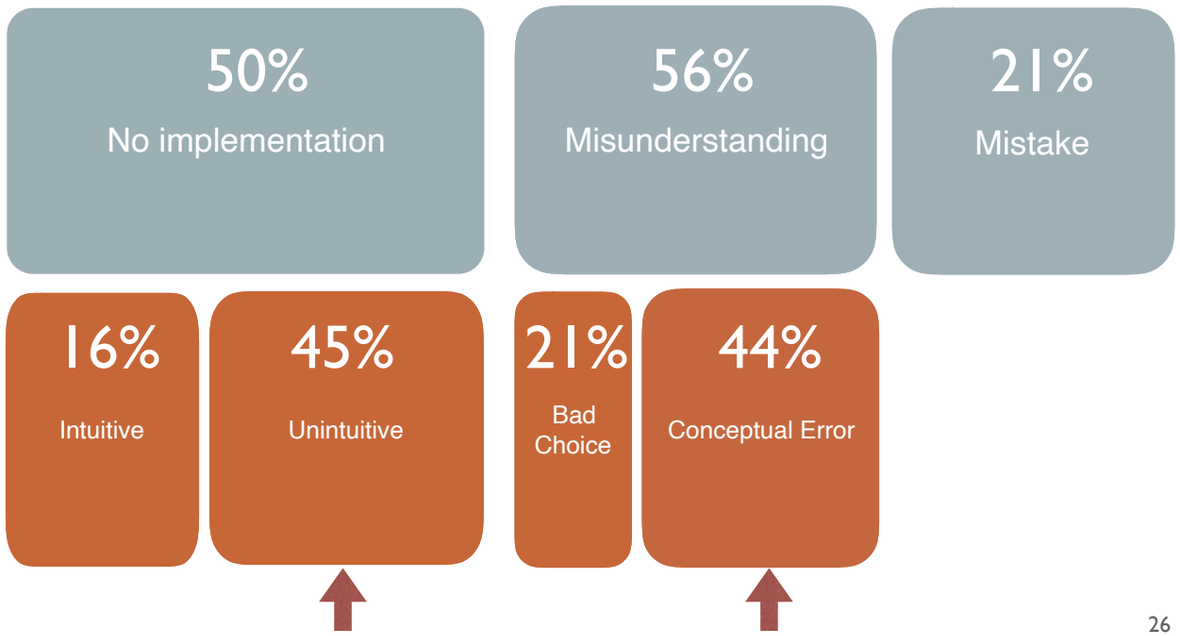
- Made a “Mistake”
  - Control flow mistake
  - Skipped algorithmic step

The final class is full of codes where teams made coding mistakes. Codes within this class are things like insufficient error checking, control flow mistakes, and teams that skipped an algorithmic step. An example of this is this team that skip checked to see if the nonce is the same as a previous nonce on line 10, but they fail to actually save any of the previous nonces, so this always returns true…. Meaning they intended to save the nonce to check it but accidentally skipped that step.



So from our coding process, we developed these classes. I’m going to explain what each of these mean later.

PREVALENCE



Most knew they needed security and picked the right tools, but didn’t know all the security requirements and how to implement them all correctly.

RECOMMENDATIONS

- Simplify API design
  - Build in security primitives and focus on common use-cases
- Indicate security impact of non-default use in API Documentation
  - Explain the negative effects of turning off certain things
- Vulnerability Analysis Tools
  - More emphasis on design-level conceptual issues

It may be useful to build in basic security primitives to API design. Can we build in the use of a MAC or nonce? Could we possibly add basic primitives like secure messaging or secure logs?

Let users know that if they choose to turn off a certain security primitive that it may cause major security vulnerabilities. Make the error messages more clear so we don't have developers googling them and then copying and pasting from Stackoverflow

It's easy to blame security education in the failure of developers to use security primitives correctly, but as we saw in this competition that definitely isn't fair. Many of the teams in our data had completed a security competition and yet they still failed to implement some things correctly. It could be that the topics presented were not emphasized and driven home in a meaningful way. The failure to get this education portion right, a failure of BIBIFI itself, serves as a valuable lesson.

SUMMARY

- Developers struggle with security concepts
  - Mostly knew they needed security and picked reasonable tools
  - Didn't know all necessary security mitigations (Unintuitive) or all the implementation details (Conceptual Error)
- Mistakes happen, but can be reduced through code review and best practices
- Improve API design, documentation, and automation to handle conceptual nuances

It may be useful to build in basic security primitives to API design. Can we build in the use of a MAC or nonce? Could we possibly add basic primitives like secure messaging or secure logs?

Let users know that if they choose to turn off a certain security primitive that it may cause major security vulnerabilities. Make the error messages more clear so we don't have developers googling them and then copying and pasting from Stackoverflow

It's easy to blame security education in the failure of developers to use security primitives correctly, but as we saw in this competition that definitely isn't fair. Many of the teams in our data had completed a security competition and yet they still failed to implement some things correctly. It could be that the topics presented were not emphasized and driven home in a meaningful way. The failure to get this education portion right, a failure of BIBIFI itself, serves as a valuable lesson.

It may be useful to build in basic security primitives to API design. Can we build in the use of a MAC or nonce? Could we possibly add basic primitives like secure messaging or secure logs?

Let users know that if they choose to turn off a certain security primitive that it may cause major security vulnerabilities. Make the error messages more clear so we don't have developers googling them and then copying and pasting from Stackoverflow

It's easy to blame security education in the failure of developers to use security primitives correctly, but as we saw in this competition that definitely isn't fair. Many of the teams in our data had completed a security competition and yet they still failed to implement some things correctly. It could be that the topics presented were not emphasized and driven home in a meaningful way. The failure to get this education portion right, a failure of BIBIFI itself, serves as a valuable lesson.

SUMMARY

Questions
dvotipka@cs.umd.edu
sec-professionals.cs.umd.edu

- Developers struggle with security concepts
  - Mostly knew they needed security and picked reasonable tools
  - Didn't know all necessary security mitigations (Unintuitive) or all the implementation details (Conceptual Error)
- Mistakes happen, but can be reduced through code review and best practices
- Improve API design, documentation, and automation to handle conceptual nuances

It may be useful to build in basic security primitives to API design. Can we build in the use of a MAC or nonce? Could we possibly add basic primitives like secure messaging or secure logs?

Let users know that if they choose to turn off a certain security primitive that it may cause major security vulnerabilities. Make the error messages more clear so we don't have developers googling them and then copying and pasting from Stackoverflow

It's easy to blame security education in the failure of developers to use security primitives correctly, but as we saw in this competition that definitely isn't fair. Many of the teams in our data had completed a security competition and yet they still failed to implement some things correctly. It could be that the topics presented were not emphasized and driven home in a meaningful way. The failure to get this education portion right, a failure of BIBIFI itself, serves as a valuable lesson.