


DePaul University



# SE480 Week 3 – Security, Testability, Interoperability



Steven Engelhardt



DePaul University

Autumn 2020

# Table of Contents

# Summary of Last Week



- *Availability* is a property of software that it is there and ready to carry out its task when the user needs it to be
- The business cost of lack of availability can be extraordinarily high
- There are many architectural tactics to help improve system availability, which can be classified into fault detection, recovery, and prevention
- *Safety* is about software's ability to avoid entering states that cause or lead to damage, injury, or loss of life

# Table of Contents

# Question 1

(1 point(s)) Which of these is the *least appropriate* definition of software architecture? Software architecture is:

1. A concrete view of the system which emphasizes implementation, algorithms, and data details.
2. The set of structures needed to reason about the system, which comprise software elements, relations among them, and properties of both.
3. The highest-level breakdown of a system into its parts; ultimately, whatever the important stuff is.
4. A bridge between business goals and the final (concrete) resulting system.

# Question 1

(1 point(s)) Which of these is the *least appropriate* definition of software architecture? Software architecture is:

1. **A concrete view of the system which emphasizes implementation, algorithms, and data details.**
2. The set of structures needed to reason about the system, which comprise software elements, relations among them, and properties of both.
3. The highest-level breakdown of a system into its parts; ultimately, whatever the important stuff is.
4. A bridge between business goals and the final (concrete) resulting system.

# Question 2



(1 point(s)) The return on investment of architecture is highest when:

1. Time horizon is short and complexity/risks are high.
2. Time horizon is short and complexity/risks are low.
3. Time horizon is long and complexity/risks are high.
4. Time horizon is long and complexity/risks are low.

# Question 2



(1 point(s)) The return on investment of architecture is highest when:

1. Time horizon is short and complexity/risks are high.
2. Time horizon is short and complexity/risks are low.
3. **Time horizon is long and complexity/risks are high.**
4. Time horizon is long and complexity/risks are low.

# Question 3

(3 point(s)) Per week 1's lecture, provide three skills required to be a good architect.

# Question 3

(3 point(s)) Per week 1's lecture, provide three skills required to be a good architect.



- Technical credibility and programming skills
- Problem decomposition skills
- Domain expertise
- Writing skills
- Business acumen
- Mentorship and consensus-building skills
- Leadership skills
- Persuasion skills
- Organizational politics awareness
- Makes decisions under uncertainty
- Optimism

# Question 4



(1 point(s)) Which of these is the most correct definition of a quality attribute? A quality attribute is:



1. A measurable or testable property of a system that is used to indicate how well the system satisfies the needs of its stakeholders.
2. A design decision with zero degrees of freedom.
3. What the system must do, and how it must behave or react to runtime stimuli.
4. Those requirements that play an important role in determining the architecture of the system.

# Question 4

(1 point(s)) Which of these is the most correct definition of a quality attribute? A quality attribute is:

1. **A measurable or testable property of a system that is used to indicate how well the system satisfies the needs of its stakeholders.**
2. A design decision with zero degrees of freedom.
3. What the system must do, and how it must behave or react to runtime stimuli.
4. Those requirements that play an important role in determining the architecture of the system.

# Question 5



(1 point(s)) What is the best term which describes "a deviation of a system from its specification, where the deviation is externally visible?"

1. Availability
2. Fault
3. Failure
4. Error

# Question 5



(1 point(s)) What is the best term which describes "a deviation of a system from its specification, where the deviation is externally visible?"

1. Availability
2. Fault
3. **Failure**
4. Error

# Question 6



(1 point(s)) Your IT department guarantees that they will respond to any Jira ticket within 12 hours of it being filed. Which term best describes this guarantee?



1. Service-level agreement (SLA)
2. Service-level objective (SLO)
3. Operational-level agreement (OLA)
4. Recovery time objective (RTO)

# Question 6



(1 point(s)) Your IT department guarantees that they will respond to any Jira ticket within 12 hours of it being filed. Which term best describes this guarantee?



1. Service-level agreement (SLA)
2. Service-level objective (SLO)
3. **Operational-level agreement (OLA)**
4. Recovery time objective (RTO)

# Question 7



(2 point(s)) Fill in the following equation which describes how to calculate availability.



$$\text{Availability } \% = \frac{\quad}{\quad} \times 100\%$$

# Question 7



(2 point(s)) Fill in the following equation which describes how to calculate availability.



$$\text{Availability } \% = \frac{\underline{\textbf{Agreed Service Time}} - \underline{\textbf{Downtime}}}{\underline{\textbf{Agreed Service Time}}} \times 100\%$$

# Question 8

(3 point(s)) A service provider measures availability over a month by dividing it into 10-minute intervals and performing a *synthetic transaction test* every interval. If the synthetic transaction test fails, the system is considered 'down' for the entire 10 minute interval. Scheduled downtime is excluded from agreed service time.
In the month of January, the service provider took one 8 hour window of scheduled downtime, and the synthetic transaction test failed 20 times outside of the scheduled downtime window. What was the availability of the system for that month? Show your work.

# Question 8

(3 point(s)) A service provider measures availability over a month by dividing it into 10-minute intervals and performing a *synthetic transaction test* every interval. If the synthetic transaction test fails, the system is considered 'down' for the entire 10 minute interval. Scheduled downtime is excluded from agreed service time.
In the month of January, the service provider took one 8 hour window of scheduled downtime, and the synthetic transaction test failed 20 times outside of the scheduled downtime window. What was the availability of the system for that month? Show your work.



$$\text{Applicable hours} = 31 \text{ days} \times \frac{24 \text{ hours}}{\text{day}} - 8 \text{ hours downtime} = 736 \text{ hours}$$

$$\text{Num intervals} = 736 \text{ hours} \times \frac{60 \text{ minutes}}{\text{hour}} \times \frac{1 \text{ interval}}{10 \text{ minutes}} = 4,416$$

$$\text{Availability} = \frac{4,416 - 20}{4,416} \times 100 = \boxed{99.547\%}$$

# Question 9

(3 point(s)) Consider a distributed data storage system which uses triple modular redundancy where the probability of an catastrophic failure is 0.2% for each of the three copies of the data. All components fail independently. What is the expected availability for the overall system?

# Question 9

(3 point(s)) Consider a distributed data storage system which uses triple modular redundancy where the probability of an catastrophic failure is 0.2% for each of the three copies of the data. All components fail independently. What is the expected availability for the overall system?





$$P_{2 \text{ failures}} = 3 \times 0.002^2 \times (1 - 0.002) = 0.000011976$$

$$P_{3 \text{ failures}} = 0.002^3 = 0.000000008$$



$$\begin{aligned} P_{\text{failure}} &= P_{2 \text{ failures}} + P_{3 \text{ failures}} \\ &= 0.000011976 + 0.000000008 \\ &= 0.000011984 = 0.0011984\% \end{aligned}$$

$$\text{Availability} = 1 - P_{\text{failure}} = \boxed{99.9988\%}$$

# Question 10



(1 point(s)) Which answer best describes the purpose of applying availability tactics?



1. To be able to reintroduce failure components back into normal operation.
2. To prevent faults from occurring.
3. To detect or anticipate faults.
4. To keep faults from becoming failures.

# Question 10



(1 point(s)) Which answer best describes the purpose of applying availability tactics?



1. To be able to reintroduce failure components back into normal operation.
2. To prevent faults from occurring.
3. To detect or anticipate faults
4. **To keep faults from becoming failures.**

# Question 11

(1 point(s)) You are attending a presentation by a software architect. In this presentation, the architect explains that their web server sends a message every 5 seconds to their monitoring system indicating that it is alive. If the monitoring system fails to receive 3 messages in a row, the monitoring system considers the web server down and marks it out of service. Which availability tactic is the architect most likely describing?

1. Ping/echo
2. Watchdog
3. Timeout
4. Heartbeat

# Question 11

(1 point(s)) You are attending a presentation by a software architect. In this presentation, the architect explains that their web server sends a message every 5 seconds to their monitoring system indicating that it is alive. If the monitoring system fails to receive 3 messages in a row, the monitoring system considers the web server down and marks it out of service. Which availability tactic is the architect most likely describing?

1. Ping/echo
2. Watchdog
3. Timeout
4. **Heartbeat**

# Question 12



(1 point(s)) You have recently designed a web application which uses blue-green deployment. Which is the *least likely* reason you chose to use blue-green deployment?



1. Reduced downtime
2. Reduced deployment risk
3. Lower cost
4. Ability to rollback

# Question 12



(1 point(s)) You have recently designed a web application which uses blue-green deployment. Which is the *least likely* reason you chose to use blue-green deployment?



1. Reduced downtime
2. Reduced deployment risk
3. **Lower cost**
4. Ability to rollback

# Question 13



(1 point(s)) Which of the below is the best definition of safety as it relates to software architecture?



1. Software must be designed to never lead to the loss of life
2. Software must never fail when failure could cause injury to a human
3. Software should avoid entering states that cause damage, and limit damage when it does enter into bad states
4. Software should never be used to ensure safety when a hardware device could do the same

# Question 13



(1 point(s)) Which of the below is the best definition of safety as it relates to software architecture?



1. Software must be designed to never lead to the loss of life
2. Software must never fail when failure could cause injury to a human
3. **Software should avoid entering states that cause damage, and limit damage when it does enter into bad states**
4. Software should never be used to ensure safety when a hardware device could do the same

# Table of Contents

# Question 1



(20 point(s)) Malan and Bredemeyer (M&B) outline several important architectural decisions that need to be made for a system. They refer to the first one as *System Priority Setting* which means that for complex systems an architect needs to decide where to excel. Based on Adrian Cockcroft's presentation, what are the areas that Netflix has decided to excel in? Explain your answer.

# Question 2



(20 point(s)) Based on M&B's notion of *System Properties and Cross-Cutting Concerns*, can you identify any cross-cutting concerns? Describe ONE of them. What evidence (reason to believe) do you have that this is cross-cutting?

# Question 3



(20 point(s)) In the past, Netflix required nearly all teams to exclusively use Java for service and component development (although there have been recent moves towards a polyglot ecosystem). Evaluate the requirement that all teams use Java from the perspective of M&B's *minimalist architecture* principle. What are the advantages and disadvantages of a monolingual ecosystem? Why do you think Netflix is moving towards polyglot development? Do you agree with their decision?

# Question 4



(20 point(s)) Netflix was an early adopter of, and continues to embrace, the public cloud (Amazon AWS), whereas some companies like Apple & Dropbox are moving away from the public cloud. List at least three advantages of using the public cloud, and three reasons of moving away from the public cloud. Do you agree with Netflix's decision to remain with the public cloud (provide detailed reasoning)?

# Question 5



(20 point(s)) What was your most important take-away (i.e. lesson learned) from Cockcroft's presentation?

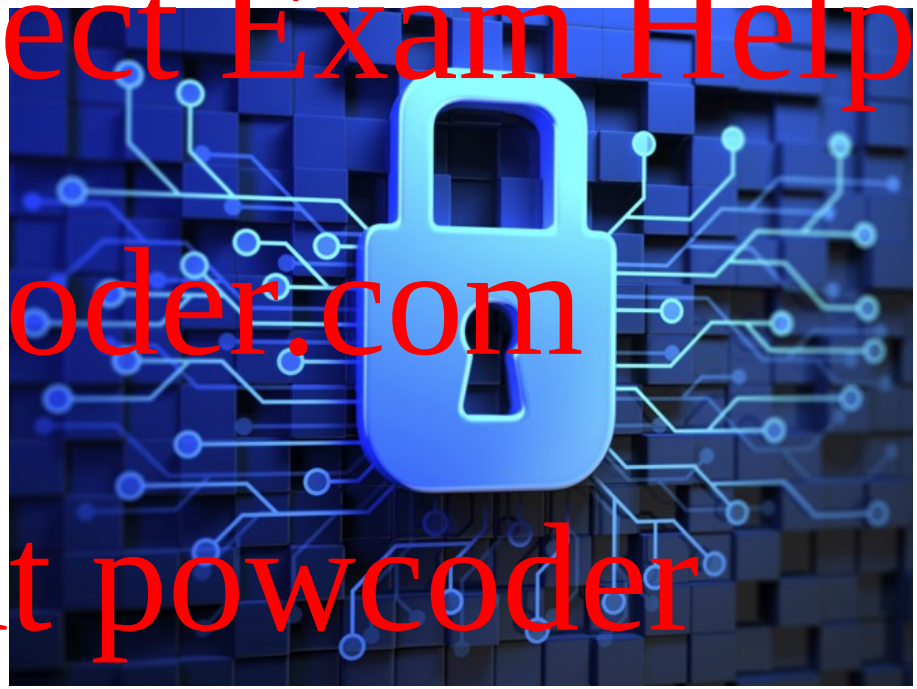
# Table of Contents

# Table of Contents

# What is Security?



- *Security* is a measure of the system's ability to protect data and information from unauthorized access while still providing access to people and systems that are authorized.



# Why is Security Important?

- In 2017, Russian military hackers released a worm called NotPetya as an act of cyberwar against Ukraine. [Gre18]

# Why is Security Important?

- In 2017, Russian military hackers released a worm called NotPetya as an act of cyberwar against Ukraine. [Gre18]
- NotPetya's goal was purely destructive. It irreversibly encrypted computers' master boot records.

# Why is Security Important?

- In 2017, Russian military hackers released a worm called NotPetya as an act of cyberwar against Ukraine. [Gre18]
- NotPetya's goal was purely destructive. It irreversibly encrypted computers' master boot records.
- The worm spread beyond the Ukraine and to Maersk, the world's largest shipping company, as well as many other companies and governments.

# Why is Security Important?

- In 2017, Russian military hackers released a worm called NotPetya as an act of cyberwar against Ukraine. [Gre18]
- NotPetya's goal was purely destructive. It irreversibly encrypted computers' master boot records.
- The worm spread beyond the Ukraine and to Maersk, the world's largest shipping company, as well as many other companies and governments.
- 17 out of Maersk's 76 terminals were shut down completely. No new bookings could be made. Ship manifests were inaccessible so ships could not be unloaded.

# Why is Security Important?

- In 2017, Russian military hackers released a worm called NotPetya as an act of cyberwar against Ukraine. [Gre18]
- NotPetya's goal was purely destructive. It irreversibly encrypted computers' master boot records.
- The worm spread beyond the Ukraine and to Maersk, the world's largest shipping company, as well as many other companies and governments.
- 17 out of Maersk's 76 terminals were shut down completely. No new bookings could be made. Ship manifests were inaccessible so ships could not be unloaded.
- Every single one of Maersk's 150 or so domain controllers was infected and was unrecoverable, except one: a Ghanian machine which had lost Internet acess due to a power blackout.

# Why is Security Important?

- In 2017, Russian military hackers released a worm called NotPetya as an act of cyberwar against Ukraine. [Gre18]
- NotPetya's goal was purely destructive. It irreversibly encrypted computers' master boot records.
- The worm spread beyond the Ukraine and to Maersk, the world's largest shipping company, as well as many other companies and governments.
- 17 out of Maersk's 76 terminals were shut down completely. No new bookings could be made. Ship manifests were inaccessible so ships could not be unloaded.
- Every single one of Maersk's 150 or so domain controllers was infected and was unrecoverable, except one: a Ghanian machine which had lost Internet acess due to a power blackout.
- Maersk had to rebuild its entire network of 4,000 servers and 45,000 PCs at a cost of $870 million.

# Why is Security Important?

- In 2017, Russian military hackers released a worm called NotPetya as an act of cyberwar against Ukraine. [Gre18]
- NotPetya's goal was purely destructive. It irreversibly encrypted computers' master boot records.
- The worm spread beyond the Ukraine and to Maersk, the world's largest shipping company, as well as many other companies and governments.
- 17 out of Maersk's 76 terminals were shut down completely. No new bookings could be made. Ship manifests were inaccessible so ships could not be unloaded.
- Every single one of Maersk's 150 or so domain controllers was infected and was unrecoverable, except one: a Ghanian machine which had lost Internet acess due to a power blackout.
- Maersk had to rebuild its entire network of 4,000 servers and 45,000 PCs at a cost of $870 million.
- The cyberattack happened on June 27, 2017. A patch which would have prevented the attack was released by Microsoft on March 14, 2017.

# Why is Security Important?

- In 2017, Russian military hackers released a worm called NotPetya as an act of cyberwar against Ukraine. [Gre18]
- NotPetya's goal was purely destructive. It irreversibly encrypted computers' master boot records.
- The worm spread beyond the Ukraine and to Maersk, the world's largest shipping company, as well as many other companies and governments.
- 17 out of Maersk's 76 terminals were shut down completely. No new bookings could be made. Ship manifests were inaccessible so ships could not be unloaded.
- Every single one of Maersk's 150 or so domain controllers was infected and was unrecoverable, except one: a Ghanian machine which had lost Internet acess due to a power blackout.
- Maersk had to rebuild its entire network of 4,000 servers and 45,000 PCs at a cost of $870 million.
- The cyberattack happened on June 27, 2017. A patch which would have prevented the attack was released by Microsoft on March 14, 2017.
- The vulnerability was originally found by the NSA, who never told Microsoft, and it was leaked in early 2017 by a mysterious group called the Shadow Brokers.

# Other Security Anecdotes

- Equifax, 2017. 145 million accounts compromised. CEO, CIO, CISO all resign. Congressional hearings occurred. [Wik20a]

# Other Security Anecdotes

- Equifax, 2017. 145 million accounts compromised. CEO, CIO, CISO all resign. Congressional hearings occurred. [Wik20a]
- Yahoo, 2016. A data breach initially estimated to affect at least 500 million users caused Verizon to request a $1 billion discount to its acquisition offer (later reduced to $300 million) [Wik20c]

# Other Security Anecdotes

- Equifax, 2017. 145 million accounts compromised. CEO, CIO, CISO all resign. Congressional hearings occurred. [Wik20a]
- Yahoo, 2016. A data breach initially estimated to affect at least 500 million users caused Verizon to request a $1 billion discount to its acquisition offer (later reduced to $300 million) [Wik20c]
- Texas, 2015. A former employee of an East Texas hospital was sentenced to 18 months in federal prison for criminal HIPAA violations. [Ald15]

# Other Security Anecdotes

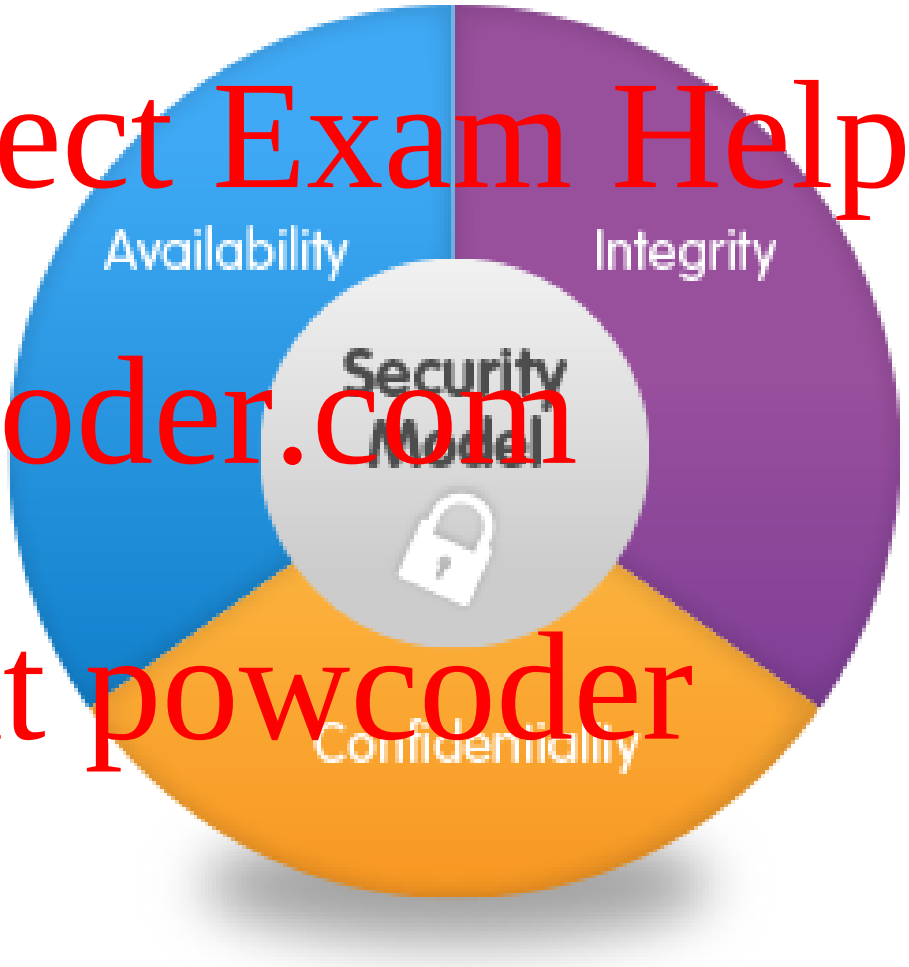
- Equifax, 2017. 145 million accounts compromised. CEO, CIO, CISO all resign. Congressional hearings occurred. [Wik20a]
- Yahoo, 2016. A data breach initially estimated to affect at least 500 million users caused Verizon to request a $1 billion discount to its acquisition offer (later reduced to $300 million) [Wik20c]
- Texas, 2015. A former employee of an East Texas hospital was sentenced to 18 months in federal prison for criminal HIPAA violations. [Ald15]
- Target, 2013. Hackers infect the company's payment-card readers. Target's profit falls 46%, and Target was later forced to pay a $18.5 million multistate settlement. [McC17]

# Other Security Anecdotes

- Equifax, 2017. 145 million accounts compromised. CEO, CIO, CISO all resign. Congressional hearings occurred. [Wik20a]
- Yahoo, 2016. A data breach initially estimated to affect at least 500 million users caused Verizon to request a $1 billion discount to its acquisition offer (later reduced to $300 million) [Wik20c]
- Texas, 2015. A former employee of an East Texas hospital was sentenced to 18 months in federal prison for criminal HIPAA violations. [Ald15]
- Target, 2013. Hackers infect the company's payment-card readers. Target's profit falls 46%, and Target was later forced to pay a $18.5 million multistate settlement. [McC17]
- Stuxnet, 2010. A malicious computer worm (suspected as an American-Israeli cyberweapon) infects Iran's industrial computer systems and causes substantial damage to Iran's nuclear program. [Wik20b]

# Characterizing Security (CIA)

- *Confidentiality* is the property that data or services are protected from unauthorized access.
- *Integrity* is the property that data or services are not subject to unauthorized manipulation.
- *Availability* is the property that the system will be available for legitimate use (i.e. is robust against denial-of-service attacks)



# Supporting CIA



- *Authentication* verifies the identities of the parties to a transaction and checks if they are truly who they claim to be.
- *Authorization* grants a user the privileges to perform a task.
- *Nonrepudiation* guarantees that the sender of a message cannot later deny having sent the message, and that the recipient cannot deny having received the message.

# Table of Contents

- Much of security is built upon the field of *cryptography*
- Cryptography is an immense and challenging field, and generally should be left to the specialists, but we architects must understand some basics

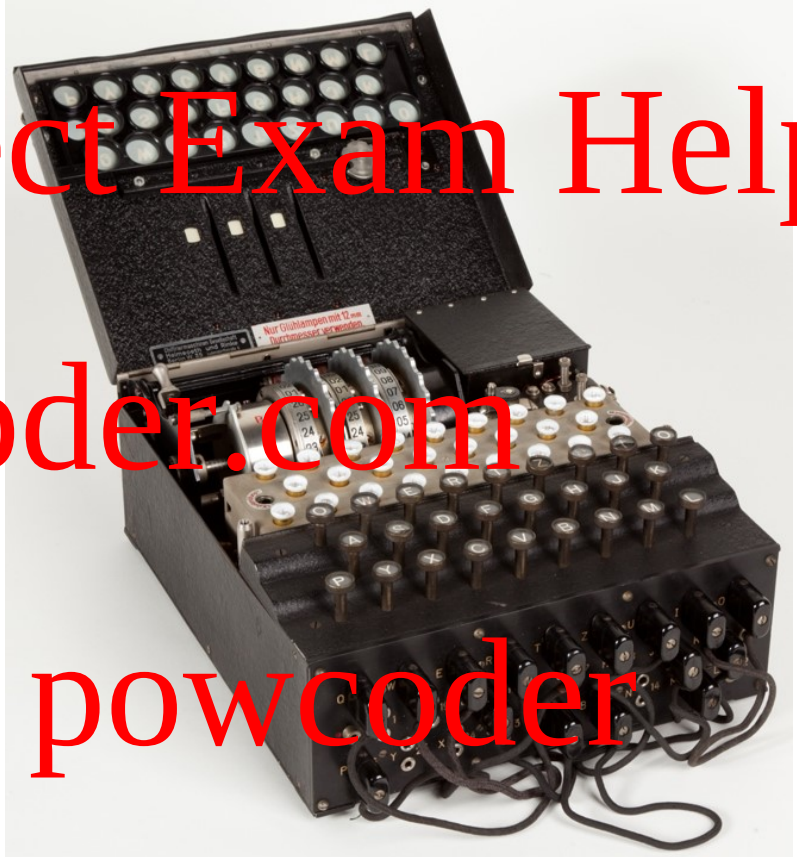


Figure: The Enigma machine, a WW2-era German rotor cipher

# Encryption



- *Encryption* is the process of encoding a message or information in such a way that only authorized parties can access it
- Encryption can be *symmetric (shared-key)* or *asymmetric (public/private-key)*



- Encryption is used to *ensure confidentiality* but it should *not* be used to ensure integrity
- Encryption is *two-way* which means it should not be used on information that should not be recoverable (e.g. passwords)!



- Example encryption algorithms include: AES, 3DES, RC5, RSA, many others

# Hashing

- A *hash function* takes an input (or message) and returns a fixed-size alphanumeric string, called the *hash value* or *message digest*
- The ideal hash function has the following properties:
  - It is extremely easy to calculate a hash for any given data.
  - It is extremely computationally difficult to calculate an alphanumeric text that has a given hash.
  - It is extremely unlikely that two slightly different messages will have the same hash.
- *Salting* is random data that is used as an additional input to a one-way function that "hashes" a password or passphrase. It ensures that two plaintext values do not hash to the same digest.
- Hash functions are particularly useful for *message integrity checks* or *user authentication*
- Example hashing algorithms include MD5, SHA-1, SHA-2, SHA-3, bcrypt, scrypt, many others

# Digital Signatures



- A *digital signature* is used to establish a sender's identity (*authentication*), that the sender cannot deny having sent the message (*non-repduation*), and that the message was not altered in transit (*data integrity*).



- A *HMAC* is a digital signature involving a *cryptographic hash function* and a *secret cryptographic key*. It is a form of *symmetric key digital signatures*.



- Examples of *assymetric key digital signatures* include *RSA* and *DSA*.

# Tokenization



- *Tokenization* is the process of substituting a sensitive data element with a non-sensitive equivalent, referred to as a token, that has no extrinsic or exploitable meaning or value.
- It's not directly related to cryptography but it is a useful technique that occasionally substitutes for a cryptographic technique



- Tokenization examples:
  - Credit card processors often use tokenization to replace real card numbers with tokens which, if stolen, cannot be used at other sites
  - Rather than storing a social security number in your database, perhaps you should replace that value with a token and keep the actual social security numbers in a separate, high-security location

# Summary



| Purpose | Technique |
|---|---|
| Ensure confidentiality (shared secret) | Symmetric encryption |
| Ensure confidentiality (public / private key) | Assymetric encryption |
| Ensure message integrity (without authentication) | Hashing |
| Ensure authentication, integrity, and non-repudiation | Digital signature |
| Protect sensitive data from accidental disclosure | Tokenization |

# Discussion



- You are designing a web application which needs to maintain a user session using a browser cookie
- Which of these techniques are appropriate for managing that cookie?
  - Encryption
  - Hashing (non-HMAC)
  - HMAC
  - Tokenization
  - A combination of the above
- What are the relative advantages and disadvantages of each approach?

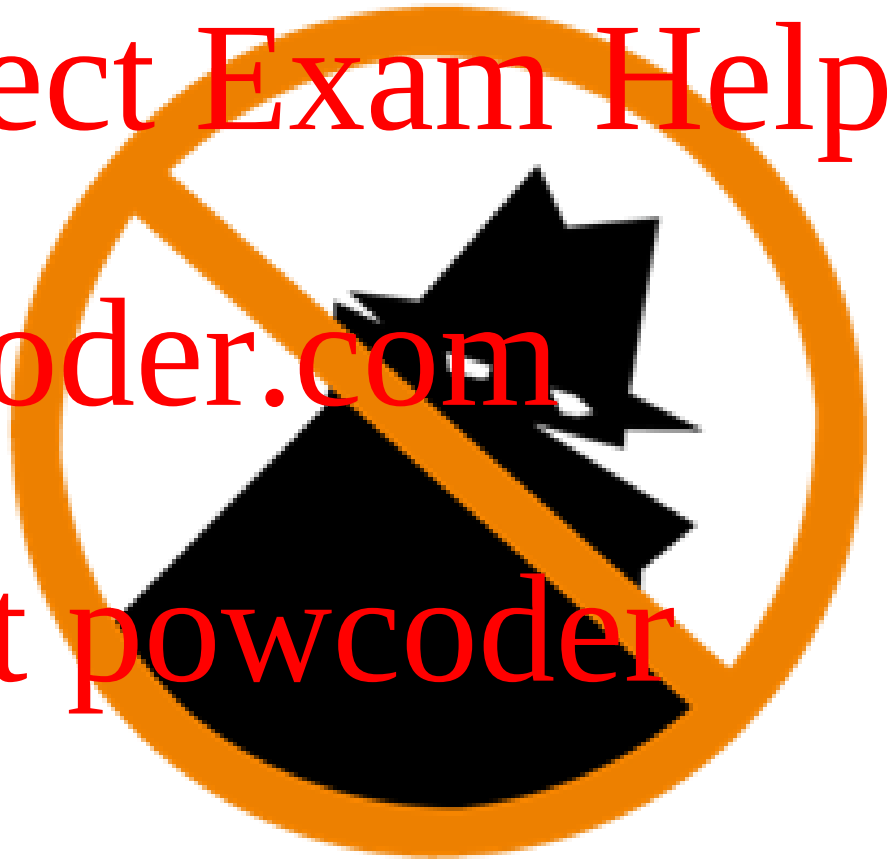
# Cryptography is Hard

- Did you realize that implementations of cryptographic algorithms need to be carefully written to *prevent timing attacks*?
- Do you know the differences between each of these algorithms and which are acceptable and which are not, and in what circumstances? DES, 3DES, MD5, AES128-ECB, AES128-CBC, AES128-CTR, PBKDF2, bcrypt, scrypt, SHA-1, SHA-2, SHA-3
- Do you know whether to *encrypt-then-hash* or *hash-then-encrypt*?
- Do you know why HMAC is implemented as `H(key || H(key || message))` not `H(key || message)`?
- Rules to live by:
  - Don't write cryptography primitives yourself
  - Always ask an expert
  - Always get security work reviewed by an expert

# Table of Contents

# Detect Intrusion

- *Detect intrusion* refers to techniques for determining that an attack is occurring or has occurred
- A common technique is the comparison of network traffic or service request patterns within a system to a set of signatures or known patterns of malicious behavior
- Often implemented using an off-the-shelf *Intrusion Detection System (IDS)*
- Average time from initial occurrence to discovery across all incident types: 66 days. [Bak18]



# Detect Service Denial



- *Detect service denial* is the comparison of the pattern or signature of network traffic coming into a system to historic profiles of known denial-of-service attacks
- Some attacks are obvious, some more subtle
- *Distributed denial of service attacks (DDoS)* are becoming more common every year, and modern ones can exceed 1 Tbps of traffic
- On October 21, 2016, an attacker leveraged a Mirai botnet consisting of 100,000 infected devices to launch a distributed denial of service attack (DDoS) attack against Dyn, a DNS provider. This caused service interruptions for many high-profile websites including Etsy, Github, Spotify, and Twitter.



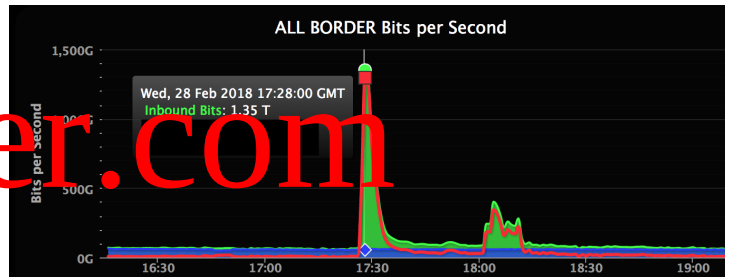



Figure: GitHub 2018 Denial of Service Attack

# Verify Message Integrity

- *Verify message integrity* refers to using techniques such as checksums or hash values to verify the integrity of messages, files, etc.
- A *man-in-the-middle attack* is where a malicious party is intercepting (and possibly modifying) messages between two systems
- Implementing integrity mechanisms (e.g. cryptographic hashes) on data sent over untrusted networks is vital to protect yourself against potential man-in-the-middle attacks
- TripWire and Verisys are commonly-used tools which constantly monitor individual hosts for file changes and reports changes to the user or system administrator




Figure: File Integrity Monitoring

# Detect Message Delay

- *Detect message delay* refers to attempting to detect potential man-in-the-middle attacks by checking the time it takes to deliver a message
- Generally speaking this is rather hard to implement, especially on systems with high jitter, and you should prefer implementing message integrity mechanisms instead.
- Optional reading: K. Benton, T. Bross. Timing Analysis of SSL/TLS Man in the Middle Attacks. 2013

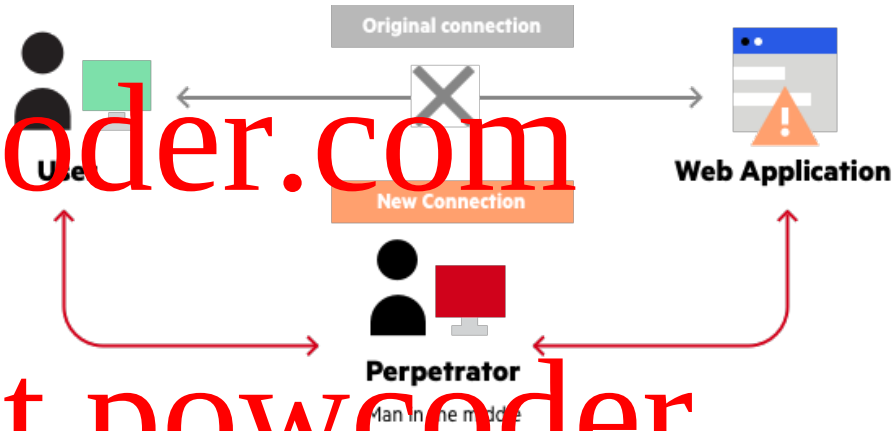



Figure: Man-in-the-Middle Attack

# Identify Actors



- *Identifying actors* is about identifying the source of any external input to the system.
- Could be by user ID, access code, IP address, protocol, etc.
- If you can't identify an actor, how do you distinguish attack traffic from normal traffic, and protect yourself from attackers?

# Authenticate Actors

- *Authentication* means that ensuring that an actor (a user or a remote computer) is actually who or what it purports to be.
- Common authentication mechanisms:
  - Passwords
  - One-time passwords
  - Digital certificates (e.g. SSL mutual auth)
  - Biometric identification
  - Single sign-on
  - Bearer tokens (e.g. those used with OAuth)
- Rule of thumb: *Authenticate everywhere*

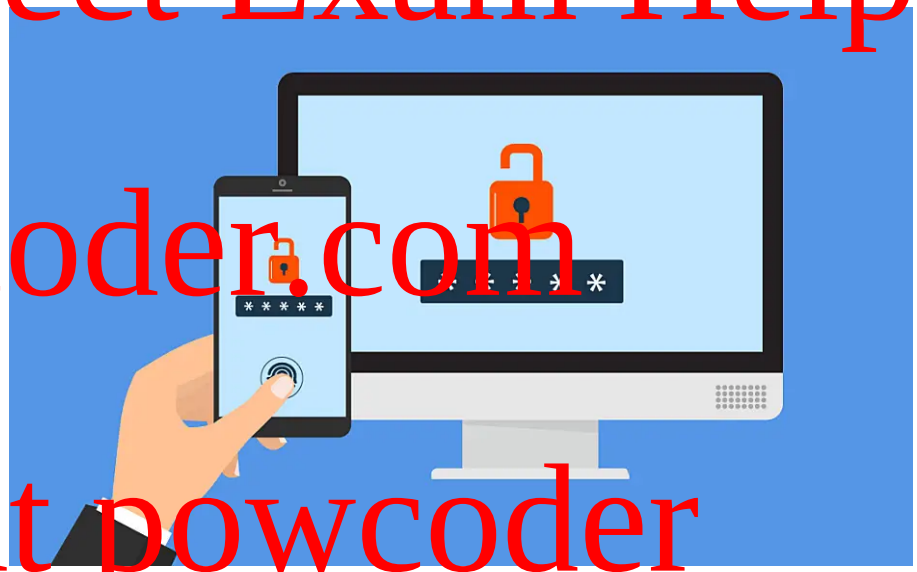
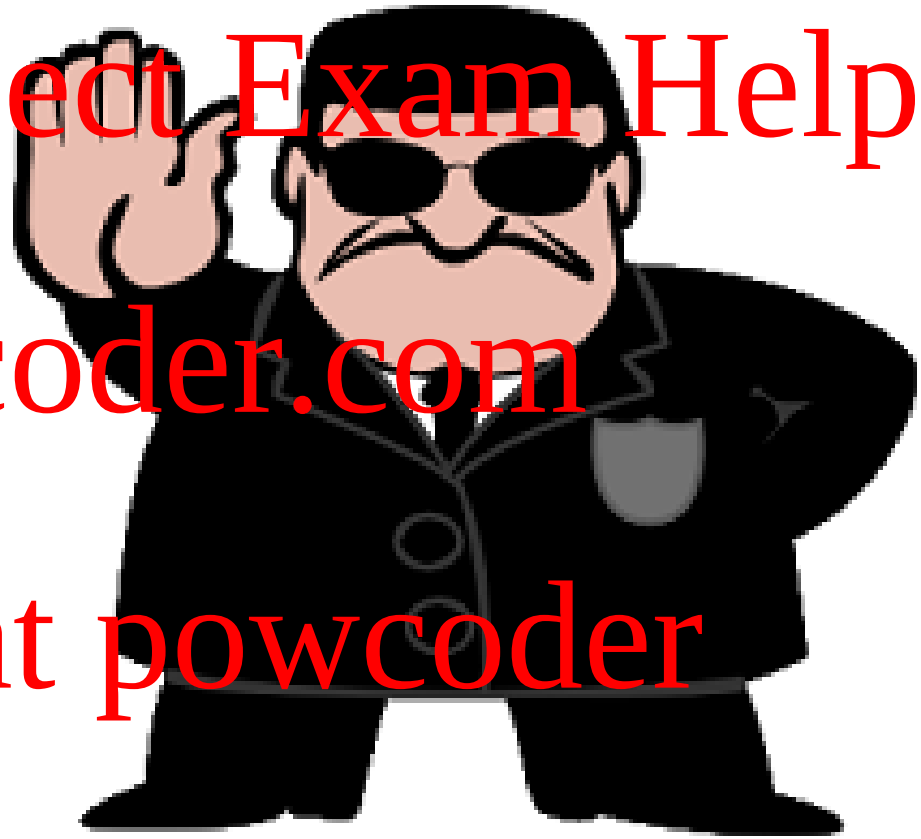


Figure: Authentication

# Authorize Actors



- *Authorization* means ensuring that an authenticated actor has the rights to access and modify either data or services.
- Common authorization models:
  - Access control lists
  - Role-based access control (RBAC)
  - Attribute-based access control (ABAC)
  - Many others

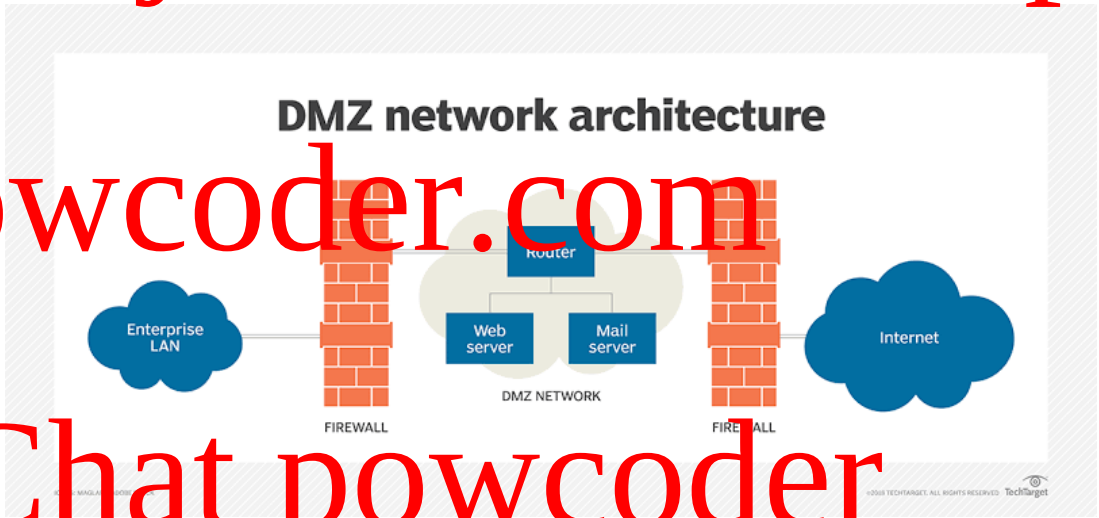


# Limit Access



- *Limiting access* involves controlling what and who may access what parts of a system.
- Common techniques:
  - Firewalls
  - Demilitarized zones (DMZs)
  - Host resource isolation (e.g. *containerization*)





# Limit Exposure



- *Limiting exposure* refers to ultimately and indirectly reducing the probability of a successful attack, or restricting the amount of potential damage
- Common techniques:
  - Security by obscurity
  - Dividing and distributing critical resources (i.e. don't put all your eggs in one basket)

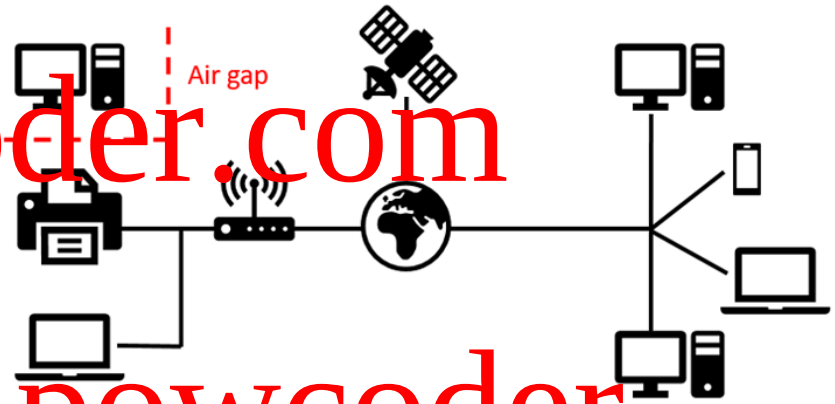
# Encrypt Data



- *Encryption* helps provide confidentiality and extra protection to persistently maintained data beyond that available from authorization
- Two types of encryption:
  - *Encryption of data at rest* – protects against information loss due to unauthorized access or theft
  - *Encryption of data in transit* – protects against eavesdropping, man-in-the-middle attacks, etc.

# Separate Entities

- *Separating entities* limits the scope of information loss in the event one system is compromised
- Common techniques:
  - Physical / virtual machine separation
  - Create *air gaps* between different portions of a system
  - Store sensitive data separately from nonsensitive data to reduce the attack possibilities from those who have access to nonsensitive data



# Change Default Settings



- *Change default settings* refers to forcing users to change default settings to prevent attackers from gaining access to the system through settings that are publicly available
- Do not ship software unauthenticated or with a default administrator password



- In 2015, Computerworld reported that there were at least 35,000 publicly accessible & insecure MongoDB databases on the Internet, collectively exposing 684.8 TB of data [Con15]
  - Either limit access (a firewall) or change default settings (use non-default password) would have prevent this

# Revoke Access



- *Revoking access* means limiting access to even normally legitimate users and uses when under attack
- This capability probably needs to be designed up front!

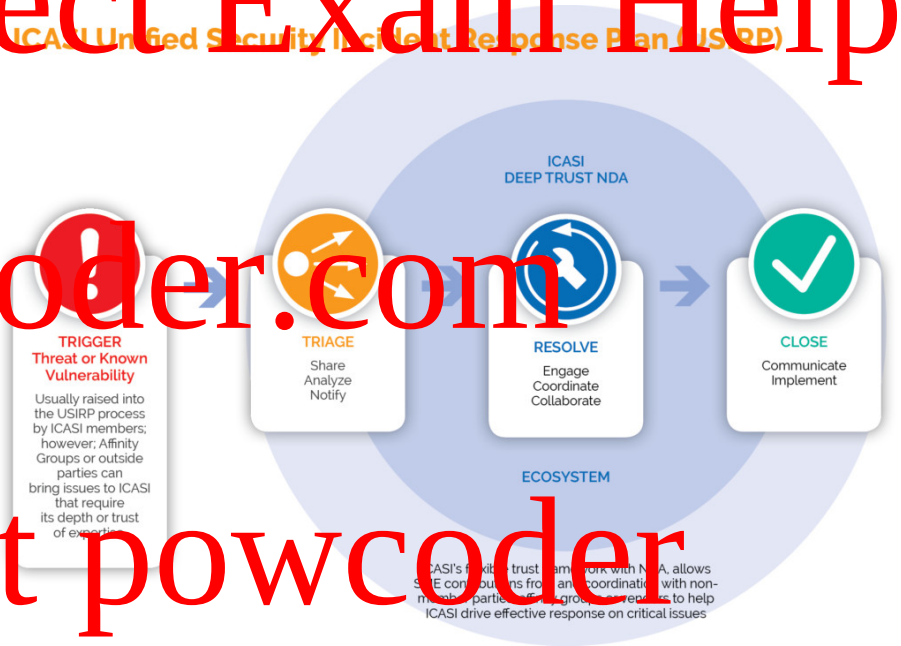
# Lock Computer

- *Lock computer* refers to limit access to an account or from a computer if there are repeated failed attempts to access an account
- Example: If you incorrectly type in your password 3 times, then your account is locked for 5 minutes
- You probably want to make sure you are limiting accesses from a specific computer, otherwise you may be opening yourself up for a denial of service attack

# Inform Actors

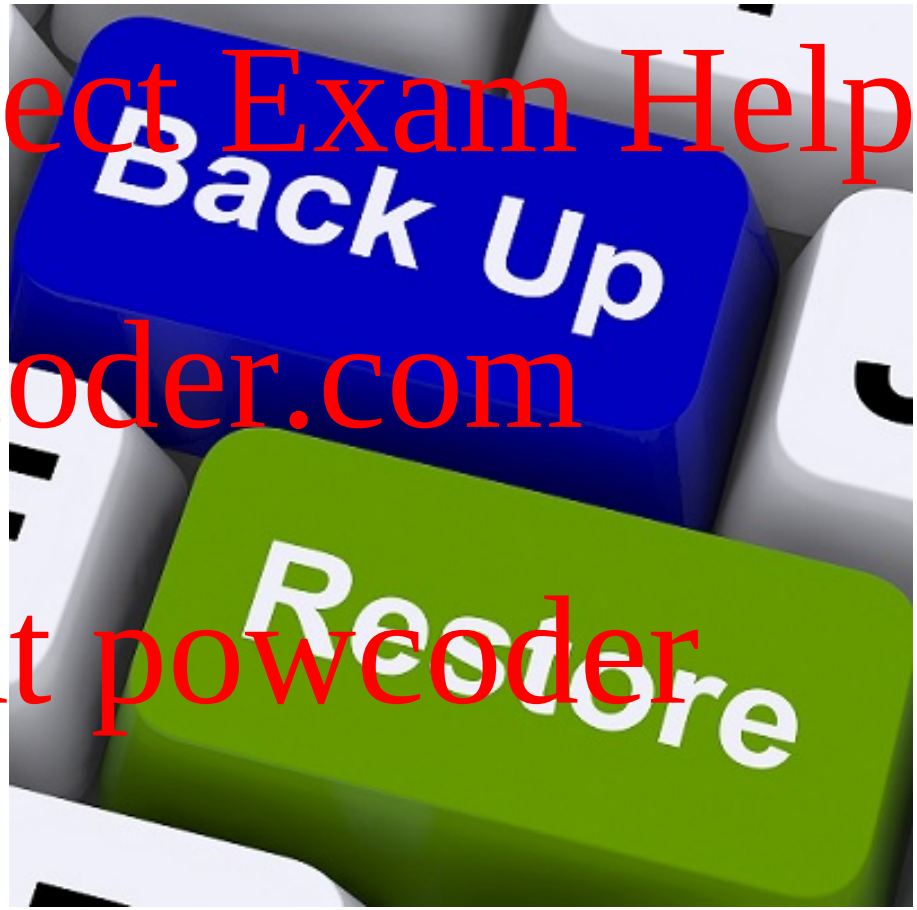


- *Inform actors* refers to notifying personnel or systems when an attack is detected
- Often driven by a prewritten *security incident response plan*
- Does your organization have one? If not, maybe you should take the lead in helping to create one.



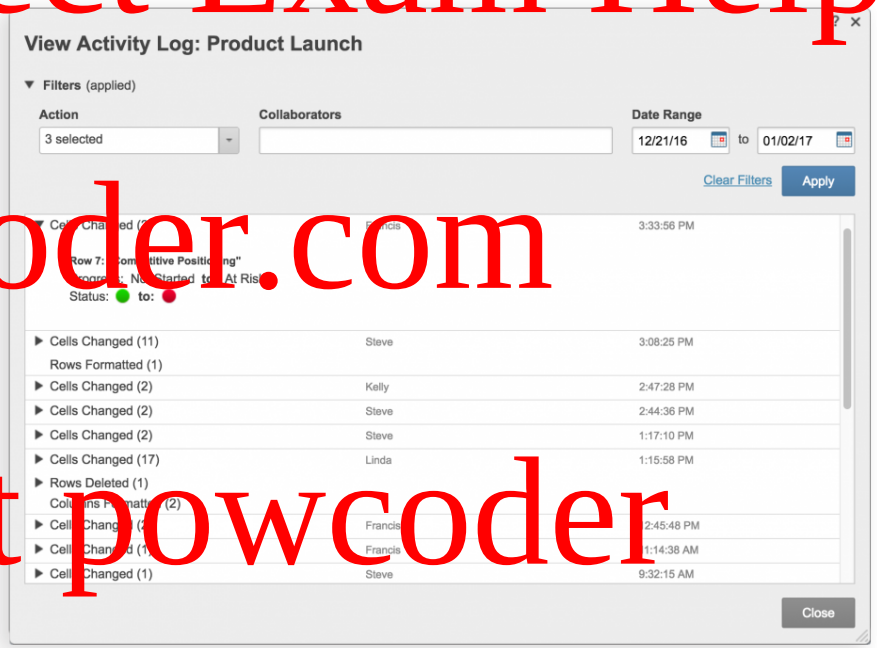
# Restore Services

- After a successful attack is resisted, we need to be able to *restore services*
- As architect, you will have to think about how you would restore services after such an attack, and design it up front.
  - Manual data cleanup?
  - Restore from backups?
- Maersk may have failed to prevent attacks, but at least they were able to (mostly) recover. Could your organization do the same?



# Maintain Audit Trail

- Not all attacks can be foreseen or prevented
- Virtually all systems should maintain an *audit trail* – a record of user and system actions and their effects – to help trace the actions of, and to identify, an attacker
- Ideally, the audit trail is built in such a way that an attacker cannot modify it in any way
- Imagine an append-only audit web service on a remote machine that is specially hardened



# Table of Contents

# OWASP



- The *Open Web Application Security Project (OWASP)* tracks the top 10 most common security vulnerabilities in web applications. They are: [OWA20]
  1. Injection
  2. Broken Authentication
  3. Sensitive Data Exposure
  4. XML External Entities (XXE)
  5. Broken Access Control
  6. Security Misconfiguration
  7. Cross Site Scripting (XSS)
  8. Insecure Deserialization
  9. Using Components with Known Vulnerabilities
  10. Insufficient Logging & Monitoring

# Architecting for Security



- The goal of security in software architecture is *not* perfect security. The goal is to *raise the cost of an attack* to an uneconomical level.
- Practice *defense in depth*. Don't rely on just one layer of security.
- *Threat modelling* is a common technique for analyzing the security of an application. It's remarkably similar to our Security Quality Scenario concept.
- It's extremely easy for a developer to inadvertently (or deliberately) introduce a *security hole* in an application.
  - People aren't perfect, and they never will be.
  - Be sure to consider *developer training*, *architectural review*, and *code review* practices as part of your organization's security posture!

# Table of Contents

# Table of Contents

# What is Testability?



- *Testability* refers to the ease with which software can be made to demonstrate its faults through (typically execution-based) testing. A testable system is one that "gives up" its faults easy.
- For a system to be properly testable, it must be possible to control each component's inputs (and possibly manipulate its internal state) and then to observe its outputs. This is often done using a *test harness*.

# Common Types of Testing

- *Unit testing* is the testing of an individual unit (module), typically done by a programmer to ensure that the module is behaving as expected
- *Integration testing* is combining a group of components together to verify that they interact correctly together
- *Functional testing* is ensuring that the specified functionality required in the system requirements works
- *Performance testing* is assessing the speed and effectiveness of the system and to make sure it is generating results within a specified time as in performance requirements
- *Stress testing* is evaluating how system behaves under unfavorable conditions (contrast to performance testing)
- *Acceptance testing* is a customer ensuring that the delivered product meets the requirements and works as the customer expected
- *Regression testing* is testing after modification of a system, component, or a group of related units to ensure that the modification is working correctly and is not damaging or imposing other modules to produce unexpected results

# Why is Testability Important?

- 30-50% of the cost of developing well-engineered systems is taken up by testing [BCK12]
- The later a bug is found, the more costly it is to fix
- Advocates of *Test-Driven Development* believe that by first turning requirements into test cases, and then improving the software to pass the new tests only, results in higher productivity and more modularized, flexible, and extensible code

# Table of Contents

# Specialized Interfaces



- Having *specialized testing interfaces* allows you to control or capture variable values for a component either through a test hardness or through normal execution
- Common examples:
  - A *report* method that returns the full state of an object
  - A *reset* method to set the internal state to a specified internal state
  - A *self-check* method that checks whether all the object's invariants are valid
  - The ability to turn on/off verbose output (very useful to be able to do this as runtime!)
- Be careful not to introduce *security vulnerabilities* here!

# Record/Playback



- *Record/playback* refers to both capturing information crossing an interface and using it as input for further testing
- Very useful, but often needs to be built up-front
- In some cases you can find tools to help you with this, e.g. tcpreplay

# Localize State Storage



- *Localize state storage* refers to keeping all state for a system/module/etc. in a single place so that it is convenient to start it in an arbitrary state for a test

# Abstract Data Sources



- *Abstract data sources* refers to abstracting the interfaces used to access a data source to let you substitute test data more easily
- Commonly implemented with the *repository pattern*
- Closely relates to *mocking* (see later slides)

# Sandbox

- *Sandboxing* refers to isolating an instance of the system from the real world to enable experimentation that is unconstrained by the worry about having to undo the consequences of the experiment
- A common technique to implement sandboxing is to *virtualize resources*
  - For example, consider virtualizing the system clock to test Daylight Savings Time boundaries

# Executable Assertions



- *Executable assertions* are (usually) hand-coded checks of method pre- and post-conditions and class-level invariants
- When an assertion triggers it means the system is in an *unexpected state*. The challenging question is what to do?
  - Abort? Retry? Ignore? Fail?

# Limit Structural Complexity



- *Limit structural complexity* refers to avoiding or resolving cyclic dependencies between components, isolating and encapsulating dependencies on the external environment, and reducing dependencies between components in general



- The *response* of a class C is a count of the number of methods of C plus the number of methods of other classes that are invoked by the methods of C
  - Keeping this metric low can increase testability



- Consider embracing *eventual consistency* rather than *immediate consistency* to keep your system simpler

# Limit Nondeterminism



- *Limit nondeterminism* means to find all sources of nondeterminism, such as unconstrained parallelism, and weeding them out as much as possible
- Nondeterministic systems are extraordinarily hard to test!

# Table of Contents

# Mocking & Dependency Injection (DI)



- *Mocking*, or using mock objects, refers to using simulated objects that mimic the behavior of real objects in controlled ways
- Mocking is a technique to help make your code easier to unit test, and is commonly seen in test-driven development
- *Mocking frameworks* make creating mock objects quick and easy
- Mocking is usually used with *dependency injection*, which is where one object supplies the dependencies of another object

# Mocking & DI Example – Step 1

```
public class FinalInvoiceStep {
  private PrinterServiceImpl printerService = null;
  private EmailServiceImpl emailService = null;

  public FinalInvoiceStep() {
    this.printerService = new PrinterServiceImpl();
    this.emailService = new EmailServiceImpl();
  }

  public void handleInvoice(Invoice invoice, Customer customer) {
    if(customer.prefersEmails()) {
      emailService.sendInvoice(invoice, customer.getEmail());
    } else {
      printerService.printInvoice(invoice);
    }
  }
}

public void testFinalInvoiceStep() {
  FinalInvoiceStep finalInvoiceStep = new FinalInvoiceStep();
  // ???
}
```

- How do I test that the class sends an email if the customer prefers an email?
- How do I test what happens if the printer service fails?

# Mocking & DI Example – Step 2

```
public class FinalInvoiceStep {
  private IPrinterService printerService = null;
  private IEmailService emailService = null;

  public FinalInvoiceStep() {
    this.printerService = new PrinterServiceImpl();
    this.emailService = new EmailServiceImpl();
  }

  public void handleInvoice(Invoice invoice, Customer customer) {
    if(customer.prefersEmails()) {
      emailService.sendInvoice(invoice, customer.getEmail());
    } else {
      printerService.printInvoice(invoice);
    }
  }
}

public void testFinalInvoiceStep() {
  FinalInvoiceStep finalInvoiceStep = new FinalInvoiceStep();
  // ???
}
```

- Have the invoice step use interfaces rather than implementations for the printer and email service
- Now I can theoretically use a test printer service, but how do I get the class to use it?

# Mocking & DI Example – Step 3

```
public class FinalInvoiceStep {
  private IPrinterService printerService = null;
  private IEmailService emailService = null;

  public FinalInvoiceStep(IPrinterService printerService,
                          IEmailService emailService) {
    this.printerService = printerService;
    this.emailService = emailService;
  }

  public void handleInvoice(Invoice invoice, Customer customer) {
    if(customer.prefersEmails()) {
      emailService.sendInvoice(invoice, customer.getEmail());
    } else {
      printerService.printInvoice(invoice);
    }
  }
}

public class TestPrinterService implements IPrinterService { ... }
public class TestEmailService implements IEmailService { ... }

public void testFinalInvoiceStep() {
  IPrinterService printerService = new TestPrinterService();
  IEmailService emailService = new TestEmailService();
  FinalInvoiceStep finalInvoiceStep =
      new FinalInvoiceStep(printerService, emailService);
  // TestPrinterService can be told to simulate a failure
}
```

- Have FinalInvoiceStep require the printer and email service in the constructor
- This is dependency injection
- Writing all these test classes is a real pain!

# Mocking & DI Example – Step 4



```
public void testFinalInvoiceStep() {
    IPrinterService printerService =
        Mockito.mock(IPrinterService.class);
    Mockito.when(printerService.printInvoice())
        .thenThrow(new PrinterOutOfServiceException());
    // ...

    FinalInvoiceStep finalInvoiceStep =
        new FinalInvoiceStep(printerService, emailService);

    finalInvoiceStep.handleInvoice(...);
    // Test to make sure the invoice class
    // behaves as expected when the printer
    // has failed
}
```



- Mocking frameworks make writing test objects easy!

# Architecting for Testability

- Keep testability in mind when designing your components
- If your component is hard to test in isolation, that may be a good sign that it has too many responsibilities and should be decomposed
- Your QA team is excellent at testing functionality, but how are they at testing non-functional requirements? Will you need to give them training, tooling?
- There are a ton of tools out there to help you out. For example, coworker of mine reported great success in using clumsy to simulate network outage scenarios. Google is your friend!
- Some things are remarkably hard to test in non production environments. For example, Facebook would be hard-pressed to simulate production levels of load on non-production servers. If you build great deployment strategies (blue-green, rollback, etc.) test-in-production might become a viable strategy. If so, then consider using:
  - *A/B testing* (a.k.a. *split testing*) is essentially an experiment where two or more variants of a page are shown to users at random, and statistical analysis is used to determine which variation performs better for a given conversion goal.
  - *Canary testing* is pushing changes to a small group of end users who are unaware that they are receiving new code

# Netflix's Simian Army – Aggressively Testing at Runtime

- *Chaos Monkey* – identifies groups of systems and randomly terminates one of the systems in a group
- *Latency Monkey* – introduces artificial delays in the client-server communication layer to simulate service degradation and measures if upstream services respond appropriately
- *Conformity Monkey* – finds instances that don't adhere to best practices and shuts them down
- *Doctor Monkey* – taps into health checks that run on each instance as well as monitors other external signs of health (e.g. CPU load) to detect unhealthy instances
- *Janitor Monkey* – searches for unused resources and disposes of them
- *Security Monkey* – finds security vulnerabilities or violations and terminates the offending instances
- *10-18 Monkey* – detects configuration and runtime problems in instances serving customers in multiple geographic regions
- *Chaos Gorilla* – simulates the outage of an entire Amazon availability zone

# Table of Contents

# Table of Contents

# What is Interoperability?



- *Interoperability* is the degree to which two or more systems can usefully exchange meaningful information via interfaces in a particular context
- Includes not only the ability to exchange data (*syntactic interoperability*) but also the ability to correctly interpret the data being exchanged (*semantic interoperability*)

# Why is Interoperability Important?



- Interoperability is what allows different systems to communicate and work together. It is the foundational idea of the Internet.
- Interoperability allows you to construct capabilities from existing systems, e.g. embed a Google Maps widget inside your mobile app
- Interoperability allows you to provide a service to used by unknown systems in the future

# How to Achieve Interoperability?



- Interoperability is achieved by having systems speak a common protocol. It is most frequently achieved by applying techniques from *service-oriented architecture (SOA)*.
- An *interface* is the set of assumptions that you can safely make about an entity. Interfaces are the key to interoperability.
- Interfaces in this context is beyond just an API (*syntax*); it includes the meaning and expectations behind the system.

# Service-Oriented Architecture



- A *service* is a collection of self-contained business functionality that exist as applications with their own design characteristics that support the strategic goals of the organization.

- *Service-orientation* is design paradigm comprised of a set of design principles that are applied to application logic that is represented as services.

- *Service-oriented architecture (SOA)* is an architectural style that supports service orientation.

- *Discovery* refers to allowing a consumer of a service to discover the location, identity, and interface of a service
- *Handling of the response*, which may be:
  - The service reports back to the requester with the response
  - The service sends its response on to another system
  - The service broadcasts its response to any interested parties

# Table of Contents

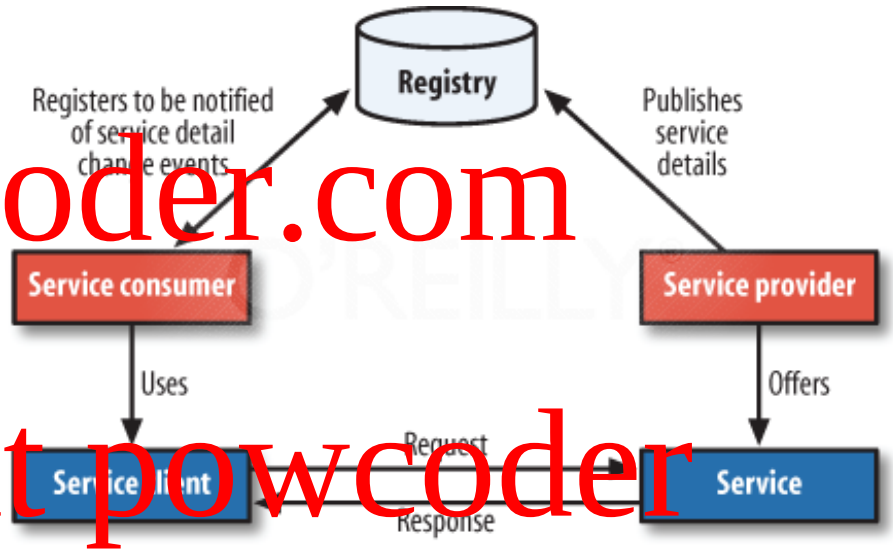
# Discover Service



- *Discover service* means to locate a service through searching a known directory service.
- By *service*, we simply mean a set of capabilities that is accessible via some kind of interface.
- The basic problem that discover service is trying to solve is "How does the client of a service make requests to a dynamically changing set of ephemeral service instances?"

# Discover Service – Service Registry

- Problem: How do clients of a service and/or routers know about the available instances of a service?
- Solution: Use a service registry, which is a database of services, their instances, and their locations. Service instances are registered with the service registry and deregistered on shutdown.
- Benefits: Discovery is enabled.
- Drawbacks: The service registry is a critical system component and must be highly available.
- Example service registries: Netflix Eureka, Apache Zookeeper, Consul, Etcd, and many implicitly-provided ones



# Service Registry Example

```
http1.json:
{
  "ID": "http1",
  "Name": "http",
  "Address": "172.17.0.3",
  "Port": 80,
  "check": {
    "http": "http://172.17.0.3:80",
    "interval": "10s",
    "timeout": "1s"
  }
}
$ curl -X PUT --data-binary @http1.json \
    http://.../v1/agent/service/register
```

Listing 1: Registering a service with Consul

```
$ curl -s http://.../v1/catalog/service/http | jq .
[
  {
    "ModifyIndex": 372,
    "CreateIndex": 372,
    "Node": "myconsul",
    "Address": "172.17.0.2",
    "ServiceID": "http1",
    "ServiceName": "http",
    "ServiceTags": [],
    "ServiceAddress": "172.17.0.3",
    "ServicePort": 80,
    "ServiceEnableTagOverride": false
  }
]
```

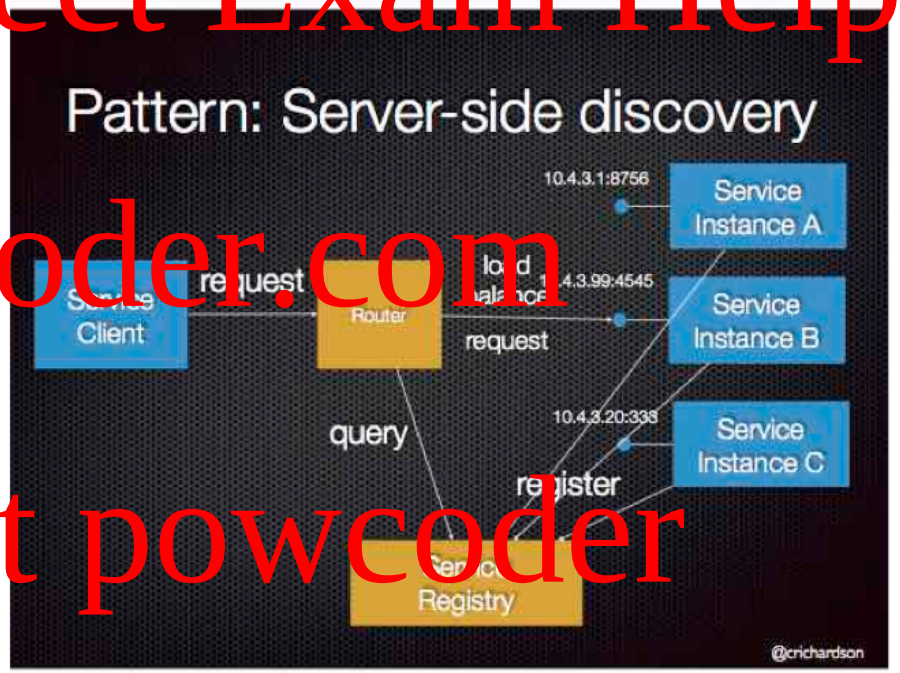
Listing 2: Discovering services with Consul

Consul [1] is an open source service registry.

[1] https://www.consul.io/

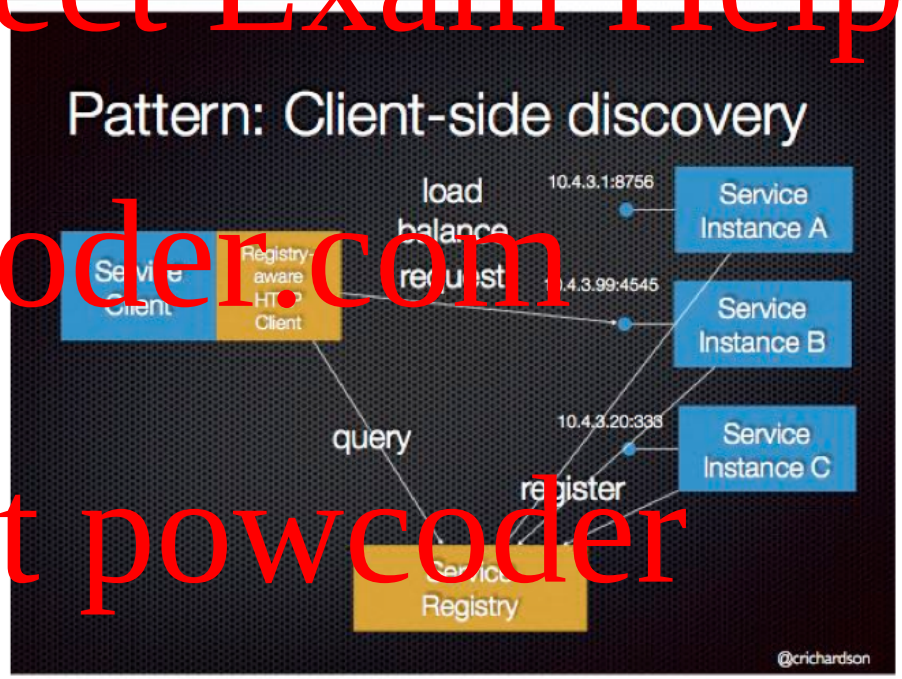
# Discover Service – Server-Side Discovery

- Server-side discovery: The client makes a request via a router (a.k.a. load balancer) that runs at a well-known location. The router queries the service registry to find available service instances.
- Benefits: Simpler client code; capability often provided by environment
- Drawbacks: Router is a critical system component and must be highly available, more network hops required
- Example server-side discovery technologies: AWS ELB, F5 BigIP, HAProxy



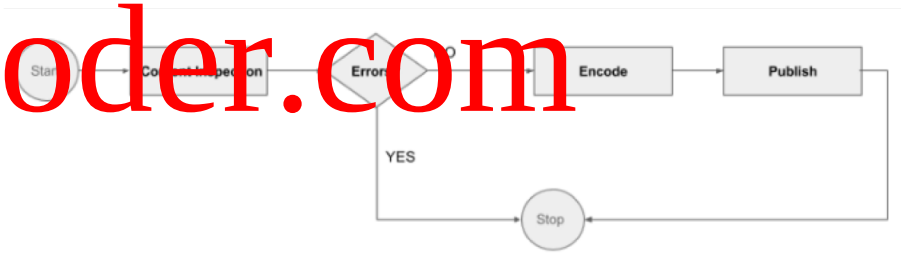
# Discover Service – Client-Side Discovery

- Client-side discovery: The client queries the service registry and makes direct connections to the service instances
- Benefits: Fewer moving parts and network hops compared to server-side discovery
- Drawbacks: Must reimplement discovery logic in each programming language/framework used by your application; clients must handle service instance failure; service instances are much harder to firewall and isolate, and frequently cannot be directly accessed over the Internet; how do you do clean shutdowns and connection draining for your service?



# Orchestrate

- To *orchestrate* means to use a control mechanism to coordinate and manage and sequence the invocation of particular services (which could be ignorant of each other). It is about mapping out workflow.
- Conceptually identical to the *mediator* design pattern
- Example technologies: BPEL, Ansible, Netflix Conductor, Mule

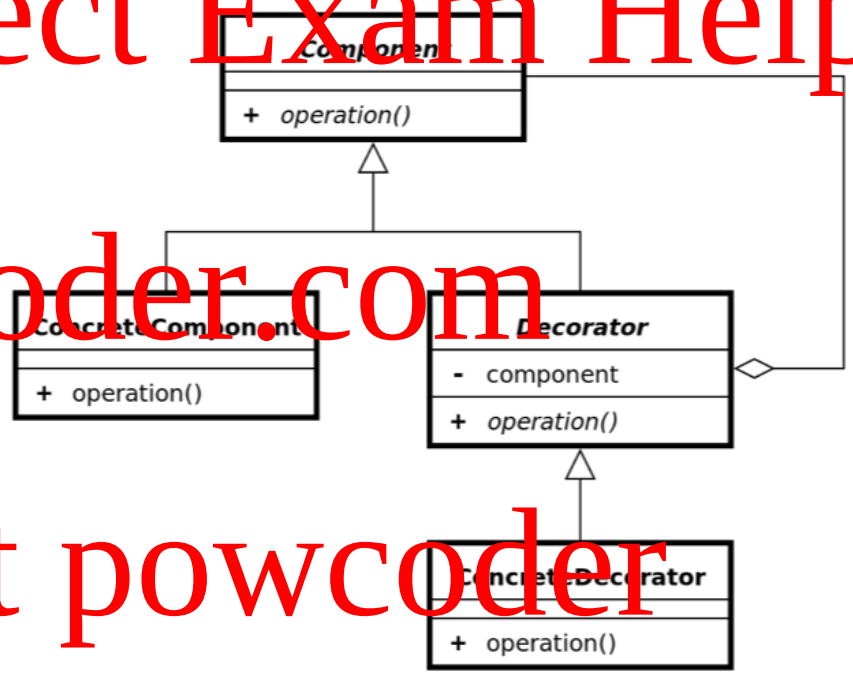


# Tailor Interface



- To *tailor interface* is to add or remove capabilities to an interface, e.g. translation, buffering, or smoothing
- Conceptually identical to the *decorator* design pattern





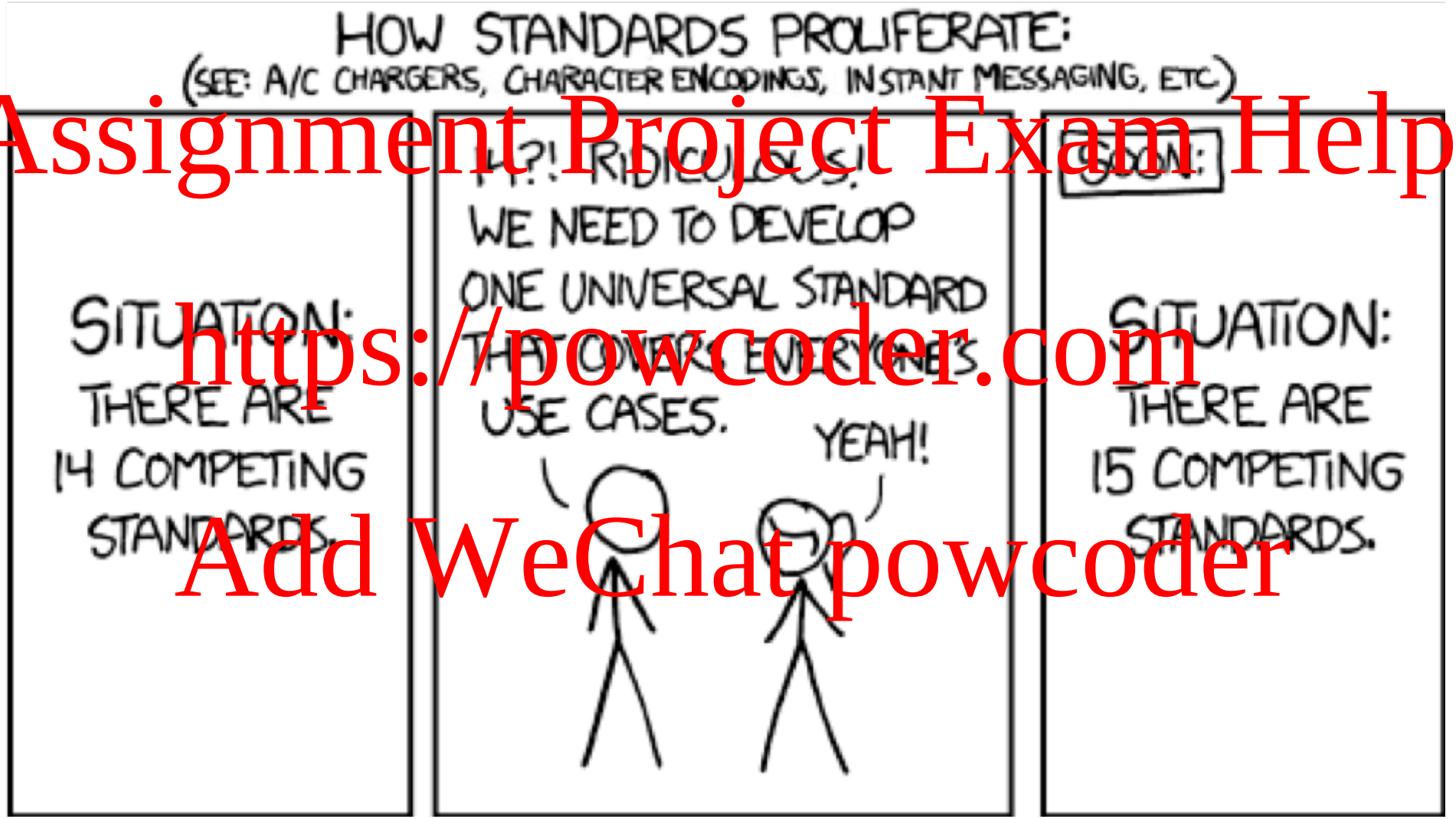
# Table of Contents

# Standards and Interoperability



- Notable standards bodies: IEEE, IETF, ISO, W3C
- Standards are *not enough* to guarantee interoperability
- Standards are incredibly numerous, inconsistent, open-ended, evolve over time, are of varying quality, and may be used as "weapons" by competing organizations
- Implementations are similarly horrible. Here's an entire RFC on observed implementation problems with TCP: https://tools.ietf.org/html/rfc2525

# Standards and Interoperability



- HTML, CSS, and JavaScript are all standardized
  - HTML 2.0 [November 1995], 3.2, 4.0, 5, 5.1 [November 2016]
  - CSS 1 [December 1996], 2, 2.1, 3 [ongoing]
  - ECMAScript 1 [June 1997], 2, ..., 8 [June 2017]
- Is it interoperable?
- Do you write to the standard or to a particular implementation?

# Table of Contents

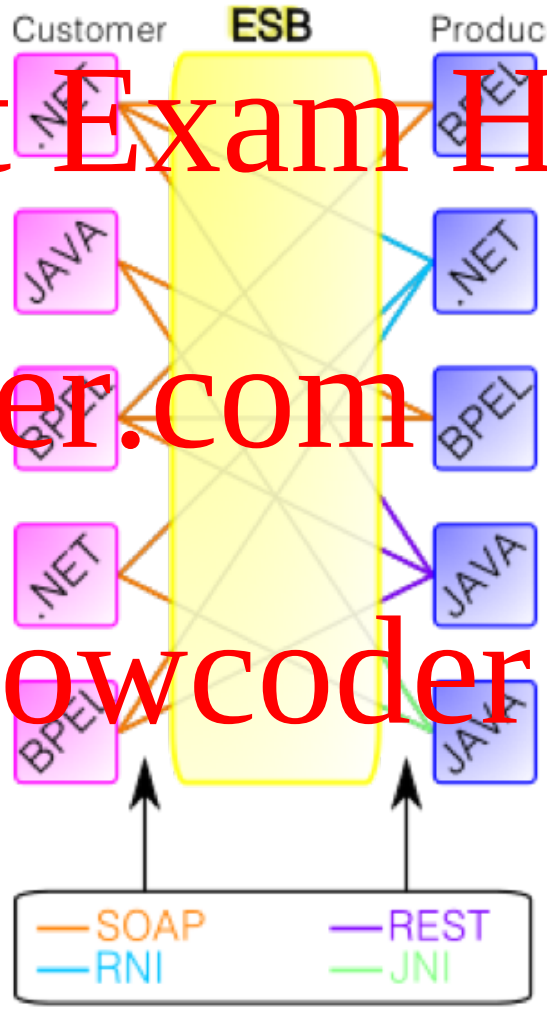
# SOA Protocols



- The two most common web-based interoperation protocols are SOAP and REST
- Both typically communicate over HTTP
- REST tends to be simpler, SOAP tends to be more "complete"
- REST seems more commonly used nowadays, but these technologies are very trendy. Remember DCOM? COBRA? XML-RPC? Perhaps GraphQL or another protocol will soon replace REST.
- When efficiency is paramount, architects sometimes choose cross-platform binary protocols like Apache Thrift

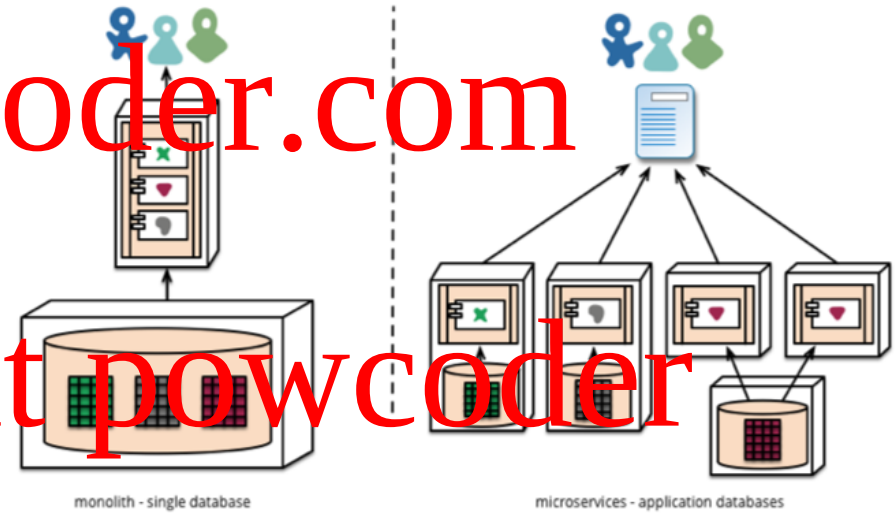
# Enterprise Service Bus

- An enterprise service bus (ESB) implements a communication system between mutually interacting software applications in a service-oriented architecture (SOA).
- The primary goal of the high-level protocol communication is enterprise application integration (EAI) of heterogeneous and complex service or application landscapes (a view from the network level).
- Includes invocation, routing, mediation, messaging, orchestration, security, etc.



# Microservices

- The microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating via lightweight mechanisms, often a HTTP resource API
- Principles
  - Built around business capabilities
  - Independently deployable
  - Smart endpoints and dumb pipes (the opposite of the ESB)
  - Product, not project
  - Highly decentralized (data management, governance, programming language, etc.)



# Architecting for Interoperability



- Embrace standards where you can, as it allows you to benefit from existing code, libraries, tools, etc.
- Ensure your interfaces are clear, well-defined, and do not leak implementation details
- When designing a service, be sure to consider:
  - RPC vs. Message vs. Resource APIs
  - Synchronous vs. asynchronous operations
  - Transactions
  - Idempotence
  - Backwards compatibility
  - Versioning
  - *Postel's law*: Be conservative in what you send, be liberal in what you accept

# Table of Contents

# Wrap-Up



- Read chapters 6,9,10 from SAIP
- Homework 2 is now available on D2L. It is due Thursday, October 8 at 5:30 PM.
- Quiz 2 will be available next week. It will cover lectures 3 and 4. It will be due Thursday, October 8 at 5:30 PM.

# Next Week



- Read chapters 8 from SAIP

- Last year, VMware released its "VMware Cloud Foundation 3.0 Architecture Poster"
- Let's read and critique! http://bit.ly/2xqj2Y5

# References



[Ald15] Steve Alder.
Hospital employee receives 18 month jail term for hipaa violations.
2015.

[Bak18] BakerHostetler.
Bakerhostetler data security incident response report demonstrates need for cyber resilience and leveraging compromise response intelligence.
2018.

[BCK12] Len Bass, Paul Clements, and Rick Kazman.
*Software Architecture in Practice*.
Addison-Wesley Professional, 3rd edition, 2012.

[Con15] Lucian Constantin.
Over 680tb of data exposed in mongodb databases.
*Computerworld*, 2015.

[Gre18] Andy Greenberg.
The untold story of notpetya, the most devastating cyberattack in history.
*Wired*, 2018.

[McC17] Kevin McCoy.
Target to pay $18.5m for 2013 data breach that affected 41 million consumers.
2017.

[OWA20] OWASP.
Owasp top ten.
2020.

[Wik20a] Wikipedia.
2017 equifax data breach.
2020.

[Wik20b] Wikipedia.
Stuxnet.
2020.

[Wik20c] Wikipedia.
Yahoo! data breaches.
2020.