CONFIDENTIAL

CONFIDENTIAL

Security Development Lifecycle

(SDLC)

Approved: Andrew Migliore

Reviewed: 2020-06-14

Overview

This document outlines policies that dictate how personal information data is accessed and

handled as we develop, debug, and triage the software we create. These policies ensure that

those who do have access understand their responsibilities, because even a secure

environment can be compromised by a single individual. Failure to properly protect sensitive

data during application development and deployment could expose the business and customers

to unacceptable consequences.

So what does sensitive data mean?

Privileged or proprietary information which, if compromised through alteration, corruption, loss,

misuse, or unauthorized disclosure, could cause serious harm to the organization or individual

owning it.

Common security problems involving sensitive data

● Cleartext storage of sensitive data

● Cleartext transmission of sensitive data

● Insecure cryptographic storage of sensitive data

● No encryption on sensitive data

● Inadequate access controls to sensitive data

Security Development Lifecycle (SDLC) Roles

Every person involved in the development of RADAR must play a role in addressing and

ensuring the security of our product and the handling of sensitive data. These roles are:

Management

● Knows that security is a reputation, customer-satisfaction, and cost-management issue.

● Understands security issues at a high level to make overall planning decisions.

● Develops security policies that communicate roles and responsibilities to the team.

RADAR — Security Development Lifecycle (SDLC) v1.3-190614--AJM 1

CONFIDENTIAL

Architect / Lead

● Owns overall technical and security design.

● Understands security concerns well to avoid exposing security exploits.

● Establishes core security principles that developers use during implementation.

● Manages awareness of security issues during design and development.

● Actively reviews and consults with the team on pull requests.

Developer

● Implements a design or function in code with security in mind.

● Fixes security vulnerabilities that result from coding errors.

● Understands secure coding best practices in detail and applies them.

QA

● Ensures security issues are caught before the application is released.

● Executes security testing (not functional testing) to find potential vulnerabilities.

● Maintains a detailed understanding of attack techniques and current threats.

Management — Understanding the business Issues

Risk

The accidental or malicious exposure of sensitive data can result in high-impact financial loss

and/or a negative impact on customer perception and business integrity. It is imperative that we

take precautions to avoid potential attacks that take advantage of the exposure of sensitive

data. For example, when passing sensitive data over a network or allowing remote access to a

storage concern containing sensitive data.

The financial impact of security breaches that result from loss of customers’ sensitive data has

become a huge reality. Financial loss is only one of the significant ways exposure of sensitive

data can negatively impact a business. Legal impact and loss of customer trust quite often

accompany the financial risks. Business decision makers should consult with their legal counsel

to evaluate the legal risks to the organization if sensitive data is leaked (see Incident Response

Policy & Plan).

Architect — Identifying risks to sensitive data

Identifying the Problem

The architect must identify and understand the nature of all information maintained by the

system, and how that information is stored and transmitted. It is also important to understand

RADAR — Security Development Lifecycle (SDLC) v1.3-190614--AJM 2

CONFIDENTIAL

the potentially sensitive nature of text used in error messages and other status information that

is delivered to an end user. Information presented in the text of error/status alerts might provide

an attacker with useful system or end user information that enables an attack.

Sensitive data requires better and more complete protection than non-sensitive data. Although

non-sensitive data should be given appropriate protections, it will require more time and

resources to ensure that security measures for sensitive data are properly implemented.

Common Attacks

Two common attack scenarios that take advantage of exposed sensitive data are *direct attacks*

to gain unauthorized access to data and *opportunistic* use of error messages or status

information to disclose paths to sensitive data. The information leaked in messages is usually

not confidential, but internal IP addresses or file system path data might be useful to an attacker

to aid further attacks.

Implementing Defenses

Implementing correct defenses to mitigate exposure of sensitive data vulnerabilities involves

first identifying how the data has been exposed, and then coming up with appropriate

mitigations to close the holes. The process steps are:

1. Build or analyze an up-to-date threat model.

2. Itemize all data stores and data flows in the threat model.

3. Determine which of these data flows and data stores could potentially expose sensitive

data.

Threat Model

When implementing defenses, developers should consult an accurate and up-to-date threat

model for the software being developed. Because all data stores and data flows in the threat

model are subject to information disclosure threats, the architect must determine which of the

data stores holds sensitive information, and those that do must be called out as requiring

special attention to make sure the data is not leaked.

RADAR — Security Development Lifecycle (SDLC) v1.3-190614--AJM 3

CONFIDENTIAL

*Figure A. example threat model for RADAR*

The threat model above shows a request flowing from the user through a browser, to the web

server and database server and how the response flows back. All requests are logged and the

system is managed by DevOps by tunneling through the bastion and using an administration

and configuration tool such as Ansible and Cloud Formation.

When determining risks, the highest risk flows should be analyzed first i.e. those that cross trust

boundaries. These are in Figure A:

● 2.0 -> 8.0 (Browser to web server)

● 8.0 -> 2.0 (Web server to browser)

● 8.0 -> 11.0 (Web server to database server)

● 11.0 -> 8.0 (Database server to web browser)

Once data stores and data flows are established, the architect will determine which of these

could potentially hold sensitive, confidential, or personal information. All data flows on a

sensitive path must be protected.

Mitigations

Information disclosure threats are mitigated with encryption and access control mechanisms e.g.

permissions and access control lists (ACLs.) For highly-sensitive data at rest, a combination of

RADAR — Security Development Lifecycle (SDLC) v1.3-190614--AJM 4

CONFIDENTIAL

permissions and encryption should be used. If possible, the system should use the

infrastructure within the operating system (OS).

Use encryption

For symmetric encryption, the Advanced Encryption Standard (AES) should be used rather than

the Data Encryption Standard (DES), 3DES, or RC4 *unless* backward-compatibility with an older

application that does not support AES is required. For RADAR this should not be the case.

Whenever possible, AES33 should be used for the encryption algorithm because of its strength

and speed. Several FIPS-approved algorithms are available for integrity checking, including

HMAC-SHA, Cipher-Based Message Authentication Code (CMAC), and Counter with Cipher

Block Chaining-Message Authentication Code (CCM).

It is always necessary to make sure that any encryption mechanism uses good key

management practices. Good key management practices should include:

● Appropriate key generation (large keys with high entropy).

● Secure key storage and protection.

● Secure key exchange.

● Key updating.

Additional information about correct key management is available in Recommendation for Key

Management – Part 1: General , published by the National Institute of Standards and

Technology (NIST).

Protecting data over a network requires the use of encryption. Like encryption for data at rest,

use existing libraries and infrastructure where possible. For data on the network, this

recommendation comes down to using Transport Layer Security (TLS).

Use existing tools

Taking advantage of reusable policy or infrastructure rather than writing new code is another

mitigation. For example, if it makes sense to do so, use the OS services to encrypt files rather

than encrypting the data with application software.

Prevent sensitive leaks in error messages

An effective way to mitigate the risk of leaking sensitive data through error messages is to

funnel all errors and warnings through a single checkpoint in the application. This single

checkpoint determines who sees what data. For example, a remote, unauthenticated user

should see very little warning information. In contrast, a local administrator should see all

information. Regardless of the user type, full error details should be written to an

administrator-readable log file. An administrator can read the full error information to help

determine why an operation failed.

RADAR — Security Development Lifecycle (SDLC) v1.3-190614--AJM 5

CONFIDENTIAL

Creating an administrator-readable log file requires applying a discretionary access control list

(DACL) to the log file with the following access control entries (ACEs):

● Everyone (Write)

● Administrators (Full Control)

A file with this DACL is often called a “drop-box” because anyone can write to the file, but only

trusted users can read the file.

Prevent sensitive leaks in email messages

Given the insecure nature of email unless using direct routing or secure solutions such as

ProtonMail, the product and development team should be cognizant of not exposing any data

via emails that might contain sensitive data even if that is not it’s intended purpose. For

example, titles and descriptions are free form text fields yet the customer might store PHI/PII

information in these fields. We should treat all such potential sensitive fields as though they are

since we cannot control the content. Email messages from our products should be scrutinized

and adjusted to account for this possibility.

Developer — Understanding the issues

The developer must make sure that the defenses designed to protect sensitive data are

appropriately implemented and do not suffer from oversights that could open the system to

attack, such as insecure key management and insecure key lengths or home-grown

cryptographic algorithms.

Examples of what not to do during implementation include:

● Using non-random encryption keys.

● Hard-coding encryption keys in application code.

● Using small encryption keys.

● Using outdated, home-grown, or insecure crypto algorithms.

Encryption keys ***must*** be protected like any other sensitive data and must be generated using

appropriately secure mechanisms. Please refer to the OpSec policies for more information.

If an application stores information in a file system or a web server or in a registry, then the

developer should consider encrypting that information.

Insecure Scrubbing

All sensitive data must be removed from memory when it is no longer needed by the application.

However, there are scenarios where some data scrubbing functions don’t work as expected.

RADAR — Security Development Lifecycle (SDLC) v1.3-190614--AJM 6

CONFIDENTIAL

Beware of compiler optimizations that may undermine this. If code scrubs secrets in memory,

the developer should call functions that are not removed by the compiler.

**Resources for developers**

● OWASP Top Ten Cheat Sheet

QA — Verifying protection of sensitive data

The challenge for QA is to find sensitive data that should be protected but is not. The tester

should consult the threat model and note the sensitive data that must be defended from

disclosure.

An effective testing technique is to add known sentinel values to the sensitive data (e.g. social

security number or credit card number like 123-45-678) then see if that data occurs in plaintext

either on the network or in storage.

In a data-driven, web-based application, a tester should add sentinel values to the test data, and

then set up a protocol analyzer to examine every packet that enters and leaves the web server

looking for the sentinel data.

Another test is to run the application through an exhaustive usability test and then scan the disk,

sector-by-sector, for the sentinel values. It is possible that the sensitive data was written to a log

file or a temporary file. By doing a sector-by-sector search, a tester can find the data if it is not

erased correctly.

Penetration testing will be performed at least annually.

**Resources for Testers**

● OWASP Testing Guide for SSL-TLS

● OWASP Testing Guide TOC

SDL

Training >> Requirements >> Design >>

Implementation >> Verification >> Release >>

Response

Our Agile SDL requires applications to document the privacy implications of the information

used or processed by the application as necessary. In such an event, the architect documents

the use of data and who should have access to the information. This is currently done in

Confluence as Solution Concepts.

RADAR — Security Development Lifecycle (SDLC) v1.3-190614--AJM 7

CONFIDENTIAL

Other methods of protecting sensitive data can be used; however, any such design must be

reviewed and approved by the OpSec team. In addition, any new design should be evaluated by

security personnel who understand cryptography well and understand how simple mistakes can

lead to exposure of sensitive information.

In order for everyone to understand their role and responsibilities, the OpSec team will hold

regular periodic training and educational meetings to go over policies and best practices e.g.

OWASP Top Ten lists.

Summary

Introducing security early in the software development life cycle, reveals problems like

inadequate data protection that can be addressed in underlying security requirements and

mitigated in design. From a financial and business perspective, it is very beneficial to eliminate

security problems as early as possible in the software development process. NIST estimates

that code fixes performed after software has been released are 30 times more costly to

complete than fixes during the design phase (see 1-10-100 rule).

The increased value of personally identifiable information (PII) and increased attacks against

software to gain access to sensitive information are clear evidence that extending the extra

effort to close potential security vulnerabilities that result from inadequate protection is critical. It

is imperative that our development team be proactive in this area and prepares to rapidly

address problems as they occur. It is equally important that our security policies and

requirements are in place when we design, implement, verify, and release code that proactively

helps protect customers from these attacks.

● For more information see RADAR Information Security Policy

● See also RADAR Security Code Review Policy

Policy Compliance

Compliance Measurement

The OpSec team will verify compliance to this policy through various methods, including but not

limited to, periodic walkthroughs, business tool reports, internal and external audits, and

feedback to the policy owner.

Training

Based on role and applicability, employees must complete policy-related training every other

year. This may include web-based training course, or instructor-led training programs.

RADAR — Security Development Lifecycle (SDLC) v1.3-190614--AJM 8

CONFIDENTIAL

Exceptions

Any exception to the policy must be approved by the OpSec team in advance.

Non-Compliance

An employee found to have violated this policy may be subject to disciplinary action, up to and

including termination of employment in addition to any civil and criminal liability. See Formal

Sanctions Policy for more details.

Maintenance

This policy will be reviewed by the Security Officer and the Privacy Officer at least once a year

or as deemed appropriate based on changes in technology, business or regulatory

requirements.

RADAR — Security Development Lifecycle (SDLC) v1.3-190614--AJM 9

Security Code Review Policy

Approved: Andrew Migliore, Neva DePalma

Reviewed: 2020-06-14

Overview

*All* product code to be released **must** be reviewed and documented either through Pull

Requests (PR) or pair programming and **must** go through a Clear to Ship meeting before being

deployed. *No exceptions* .

For information on managing Pull Requests see:

https://help.github.com/articles/using-pull-requests/

Review process

As a reviewer you should know the types of security issues that are common for the application

(e.g. web application) you are reviewing as well as any specific code changes that should be

reviewed. You should consider the following categories to determine the focus of the review:

● SQL injection

● Cross-site scripting

● Input/data validation

● Authentication

● Authorization

● Sensitive data

● Code access security

● Exception management

● Data access

● Cryptography

● Unsafe and unmanaged code use

● Configuration

● Threading

● Undocumented public interfaces

(see also our SDLC and Threat Model, and the top “ten” lists from OWASP and SANS)

Determine what types of issues you are looking for. For example, consider the following:

● General issues that affect confidentiality, integrity, and availability.

● Issues related to the application's security quality of service requirements.

● Issues related to the application's compliance requirements.

RADAR — Security Code Review Policy v1.6-2020-06-14-AJM 1

CONFIDENTIAL

● Issues related to the technologies that the application uses.

● Issues related to the functionality the application exposes.

Consider the questions

● Which common coding errors apply to the code you are reviewing?

● What are the technologies used in the application?

○ Is there a database?

○ Does the component present user defined data on a Web page?

○ Does the component interact with native code or other libraries?

○ Do users supply input to the component, either directly or through an

intermediary?

● Which of the identified threats from the threat model (see SDLC) apply to the code you

are reviewing?

After you determine what applies, you can identify areas for which the risk has *not* been

mitigated.

Determine objectives

Examples of security code review objectives:

● Make sure that all untrusted input to the component is passed to a validation routine

before it is used.

● Check error handling to make sure that exceptions are caught consistently and caught

close to their source.

● Check calculations whose results are used for memory allocation or buffer access for

numeric overflow or underflow.

● Check cryptographic routines to make sure secrets are cleared quickly.

Review code with specific goals, time limits, and knowledge of the issues you want to uncover.

Tool assisted scans

All code should be evaluated with static source analysis (SSA) tools.

First perform a scan of the code to find an initial set of issues and to discover hot spots where

additional security issues are likely to be discovered in later steps.

Static analyzers tend to be good at finding careless code practices, such as missing error

handlers, empty catch blocks, integer overflows, and scoping problems.

If you are unable to use a static analysis tool, you can perform text searches (e.g. using *grep* or

*find* on unix) on the code base looking for common patterns.

RADAR — Security Code Review Policy v1.6-2020-06-14-AJM 2

CONFIDENTIAL

Note that analysis tools frequently find false positives but on the other hand do not to develop a

false sense of security if an automated scan shows no issues in the code. This does not mean

that the code is free of vulnerabilities!

For example for the Go language, we currently use *Meta Linter* that aggregates a large set of

SSA tools that can assist with finding problems.

Supported tools:

● **go vet** - Reports potential errors that otherwise compile.

● **go vet --shadow** - Reports variables that may have been unintentionally shadowed.

● **gotype** - Syntactic and semantic analysis similar to the Go compiler.

● **deadcode** - Finds unused code.

● **gocyclo** - Computes the cyclomatic complexity of functions.

● **golint** - Google's (mostly stylistic) linter.

● **varcheck** - Find unused global variables and constants.

● **structcheck** - Find unused struct fields.

● **aligncheck** - Warn about un-optimally aligned structures.

● **errcheck** - Check that error return values are used.

● **dupl** - Reports potentially duplicated code.

● **ineffassign** - Detect when assignments to existing variables are not used.

● **interfacer** - Suggest narrower interfaces that can be used.

● **unconvert** - Detect redundant type conversions.

● **testify** - Show location of failed testify assertions (disabled by default).

● **test** - Show location of test failures from the stdlib testing module (disabled by default).

● **gofmt -s** - Checks if the code is properly formatted and could not be further simplified.

● **goimports** - Checks missing or unreferenced package imports.

Manual scans

A reviewer should complete a manual scan of the code to better understand and to recognize

patterns. This should be a quick walk through that takes no more than 10 percent of the review

time. In particular, you should review the code with the following questions in mind:

● Input data validation.

○ Does the application have an input validation architecture?

○ Is validation performed on the client, on the server, or both?

○ Is there a centralized validation mechanism, or are validation routines spread

through the code base?

● Does the application authenticate or authorize users?

● Does the code isolates separate accounts during queries?

● What roles are allowed and how do they interact?

● Is there custom authentication or authorization code?

● Error handling code. Is there a consistent error handling architecture?

RADAR — Security Code Review Policy v1.6-2020-06-14-AJM 3

CONFIDENTIAL

○ Does the application catch and throw structured exceptions?

○ Are there areas of the code with especially dense or sparse error handling?

● Complex code. Are there areas of the code that appear especially complex?

● Cryptography. Does the application use cryptography?

The result should be a set of areas that deserve further analysis.

Review code for security issues

Look for common security vulnerabilities. Trace paths through the code that are most likely to

reveal security issues. Use a question-driven approach in conjunction with other techniques

such as control flow and dataflow analysis.

● **Control flow analysis**

Control flow analysis is the mechanism used to step through logical conditions in the

code. The process is:

○ Examine a function and determine each branch condition. These can include

loops, switch statements, if statements, and try/catch blocks.

○ Understand the conditions under which each block will execute.

○ Move to the next function and repeat.

● **Dataflow analysis**

Dataflow analysis is the mechanism used to trace data from the points of input to the

points of output. Because there can be many data flows in an application, use the code

review objectives to focus the work. The process is:

○ For each input location, determine how much you trust the source of input.

○ Trace the flow of data to each possible output.

■ Note where there is data validation.

○ Move to the next input and continue.

○ Review input and output sources

■ Public interfaces

■ User interface

■ Database interaction

■ Socket interaction

■ File I/O

■ Pipes

Note: Prioritize areas where the code crosses trust boundaries.

Trust Boundaries

It can be difficult to determine how much you trust each input source. The code should not trust

input that comes from outside its component, and should fully validate all data. For performance

and maintainability reasons this may not always be practical.

RADAR — Security Code Review Policy v1.6-2020-06-14-AJM 4

CONFIDENTIAL

High trust:

● Input from code you are reviewing inside the component.

● Input that comes from known good, strongly named, managed or signed/hashed

libraries.

● Input from a database that is used only by the component and that contains data which

you can prove has been properly validated and is therefore correct by construction.

● Network data that has been signed by a known good source and is protected by IPSec

or TLS.

Medium trust:

● Input from known good libraries that have not been strongly named or signed, but are

local to the server and vendored.

● Input from a public interface that should only be accessible to trusted users.

● Input from a user interface component that should only be accessible to trusted users.

● Network data that should not be accessible to an untrusted user, such as a segmented

LAN internal to the datacenter.

Low trust:

● Input that comes from libraries that have not been strongly named or signed and are

located on the client.

● Input that comes from client code.

● Input that comes over the network.

● Input that comes from a file.

● Input that comes from a public interface that is accessible to any user.

● Input that comes from the user interface component that is accessible to any user.

● Input that comes from a database that is shared with other applications.

Carefully examine the code during the review to make sure that input validation is performed

rigorously on low-trust input and performed adequately on medium-trust input.

Pay attention to areas where the data is parsed and may go to multiple output locations. Pay

attention to intermediary output locations. Trace data back to its source, and assign trust based

on the weakest link.

Hotspots to look for

● **SQL injection**

A SQL injection attack occurs when untrusted input can modify the semantics of a SQL

query in unexpected ways. As you review the code, make sure that the SQL queries are

parameterized and that any input used in a SQL query is validated.

● **Cross-site scripting**

Cross-site scripting occurs when an attacker manages to inject script code into an

RADAR — Security Code Review Policy v1.6-2020-06-14-AJM 5

CONFIDENTIAL

application so that script code is echoed back and executed in the security context of the

application. This can allow an attacker to steal user information, including forms data and

cookies. This vulnerability can be present whenever a Web application echoes unfiltered

user input back to Web content.

● **Data access**

Look for improper storage of database connection strings and proper use of

authentication to the database.

● **Input/data validation**

Look for client-side validation that is not backed by server-side validation, poor validation

techniques, and reliance on file names or other insecure mechanisms to make security

decisions.

● **Authentication**

Look for weak passwords, clear-text credentials, overly long sessions, and other

common authentication problems.

● **Authorization**

Look for failure to limit database access, inadequate separation of privileges, and other

common authorization problems.

● **Sensitive data**

Look for mismanagement of sensitive data by disclosing secrets in error messages,

code, memory, files, or the network.

● **Unsafe code**

Look for potential buffer overflows, array out of bound errors, integer underflow and

overflow, as well as data truncation errors. Unit tests and compilation can help catch

these type of issues early.

● **Hard-coded secrets**

● Look for hard-coded secrets in code by looking for variable names such as "key",

"password", "pwd", "secret", "hash", and "salt".

● **Poor error handling**

Look for functions with missing error handlers or empty catch blocks.

● **Code that uses cryptography**

Check for failure to clear secrets as well as improper use of the cryptography APIs

themselves.

● **Undocumented public interfaces**

Undocumented interfaces should not be in our code, they are almost never given the

same level of design and test scrutiny as other code.

● **Threading or concurrency problems**

Check for race conditions and deadlocks, especially in static methods and constructors.

Continuous improvement

We do not want to repeat the same mistakes, when a flaw is detected the Lead Developer

should perform a root-cause analysis, and adjust processes, tools, or skills to avoid recurrence

RADAR — Security Code Review Policy v1.6-2020-06-14-AJM 6

CONFIDENTIAL

of the problem in the future. Training the whole team on these issues is critical to prevent issues

in the future.

Security awareness training/education

The OpSec team will hold regular periodic training and educational meetings to go over policies

and best practices e.g. OWASP Top Ten lists. The Architect and dev leads are responsible to

provide oversight during pull requests for the rest of the development team.

Policy Compliance

Compliance Measurement

The OpSec team will verify compliance to this policy through various methods, including but not

limited to, auditing the pull requests, and feedback to the policy owner.

Training

Based on role and applicability, employees must complete policy-related training every other

year. This may include web-based training courses, or instructor-led training programs.

Exceptions

Any exception to the policy must be approved by the OpSec team in advance.

Non-Compliance

An employee found to have violated this policy explicitly by attempting to merge in code without

going through the review process may be subject to disciplinary action, up to and including

termination of employment. See Formal Sanctions Policy for more details.

Maintenance

This policy will be reviewed by the Security Officer and the Privacy Officer at least once a year

or as deemed appropriate based on changes in technology, business or regulatory

requirements.

RADAR — Security Code Review Policy v1.6-2020-06-14-AJM 7

CONFIDENTIAL

Infrastructure as Code (IaC) &

Patch Management Policy

Approved: Andrew Migliore, Neva Depalma

Updated: 2020-02-27

Overview

As a software engineering organization we have adopted the twelve-factor app methodology for

building software-as-a-service apps that use declarative formats for automation and are suitable

for deployment on modern cloud platforms, thus obviating the need for traditional manually

managed servers and systems administration (see https://12factor.net/ ).

We have also chosen AWS as our cloud platform and use hardened base machine images to

stand up new compute instances.

Purpose

This document provides the processes and necessary guidelines to:

● Maintain the integrity of network systems and data by creating base machine images

(using AWS AMIs, Ansible and other IaC tools) that have been updated with the latest

OS and application security patches in a timely manner as defined below.

● Establish a baseline methodology and timeframe for patching and confirming appropriate

patch-management compliance.

Scope

The processes addressed in this document affect all servers in production managed by the

DevOps team.

Common Vulnerability Scoring System

**Rating CVSS Score Timeframe to Patch (days)**

Low 0.1 - 3.9 <= 180

Medium 4.0 - 6.9 <= 90

High 7.0 - 8.9 <= 21

Critical 9.0 - 10.0 <= 3

RADAR — Infrastructure Buildout & Patch Management Policy v1.5-20200227-AJM 1

CONFIDENTIAL

Process

1. Patches will be assessed by the OpSec team based on the severity and risk of the

vulnerabilities addressed by the patch

2. Patches will be deployed by the DevOps team once changes have been approved.

3. Always apply patches from a trusted source.

a. This is often from the OS specific software package management tool e.g. apt or

yum.

4. Test patches when necessary to identify any adverse effects.

5. Create pull request for any Ansible or AMI changes.

6. Deploy patches

a. Patches are to be deployed, in general, as soon as they are released based on

the criticality (often Low and Medium scored patches have already been applied

by AWS in the base AMI) but no later than the schedule stated above.

b. Out-of-Band security patches should be deployed ASAP to production systems

for Critical and High CVEs.

7. Exceptions

a. Systems or applications that cannot be patched to resolve a known vulnerability

must have the justification documented by the OpSec team, and necessary

security controls will be implemented to mitigate the vulnerability until the system

can be patched.

b. Systems that transmit or store protected data and cannot be patched (e.g. there

is no patch available yet) will be brought to the attention of the Privacy Manager.

Necessary security controls to compensate for the vulnerability will be

implemented by the OpSec / DevOps teams.

Patch-Compliance

The OpSec team will analyze various reporting tools when determining patch compliance.

1. All members of the OpSec team will monitor the CVE and NVD databases for relevant

security bulletins.

2. OWASP Zap and Tenable Nessus will be used in conjunction with other tools to

determine externally exposed vulnerabilities.

3. GitHub’s Security Alert feature will be configured to be *always* on in order to notify the

whole product team of known vulnerabilities.

4. Periodic vulnerability assessments will be performed by running vulnerability scans at

least monthly and hiring external reputable third party for annual penetration testing.

RADAR — Infrastructure Buildout & Patch Management Policy v1.5-20200227-AJM 2

CONFIDENTIAL

Resources

Please refer to the online Confluence documentation for How-Tos for baking base Linux AMIs.

Policy Compliance

Compliance Measurement

The OpSec team will verify compliance to this policy through various methods, including but not

limited to, periodic walkthroughs, business tool reports, internal and external audits, and

feedback to the policy owner.

Training

Based on role and applicability, employees must complete policy-related training every other

year. This may include web-based training course, or instructor-led training programs.

Exceptions

Any exception to the policy must be approved by the OpSec team in advance.

Non-Compliance

An employee found to have violated this policy may be subject to disciplinary action, up to and

including termination of employment in addition to any civil and criminal liability. See Formal

Sanctions Policy for more details.

Maintenance

This policy will be reviewed by the Security Officer and the Privacy Officer at least once a year

or as deemed appropriate based on changes in technology, business or regulatory

requirements.

RADAR — Infrastructure Buildout & Patch Management Policy v1.5-20200227-AJM 3

CONFIDENTIAL

DDoS Risk Mitigation Policy

Approved: Andrew Migliore, Neva DePalma

Reviewed: 2020-06-25

Purpose

Key considerations in the mitigation of any volumetric DDoS attacks must include the availability

of transit capacity (and diversity) and protecting resources against attack traffic.

RADAR’s main risk mitigation strategy against infrastructure and application layer DDoS attacks

is to rely on AWS services for resiliency.

AWS

The AWS infrastructure is DDoS-resilient by design and is supported by DDoS mitigation

systems that can automatically detect and filter excess traffic. RADAR has been implemented

and deployed upon these services to take advantage of these capabilities to handle attacks.

The following is a summary of the AWS services and their capabilities:

RADAR DDoS Risk Mitigation v1.2-2020-06-25-AJM 1

CONFIDENTIAL

Resiliency Strategies

Services used by RADAR within AWS Regions, like Elastic Load Balancing (ELB) and Elastic

Compute Cloud (EC2) allow RADAR to scale (vertically and horizontally) to handle unexpected

volumes of traffic within a given region.

Load Balancing

Larger DDoS attacks can exceed the resources of a single EC2 instance. To mitigate these

attacks, we use ELB to automatically scale out horizontally (based on an ELB policy) by

distributing traffic across many backend instances to manage larger volumes of unanticipated

traffic.

In addition, ELB accepts only well-formed TCP connections which means that many common

DDoS attacks, like SYN floods or UDP reflection attacks will not be accepted by ELB and will

not be passed on to the RADAR’s application layers.

Instance Size

RADAR can also scale vertically by using larger instances that:

● Support 10 Gigabit network interfaces

Each instance is therefore able to support a larger volume of traffic helping prevent

interface congestion for any traffic that has managed to reach application EC2 instances.

● Support Enhanced Networking Support

Each instance therefore has higher I/O performance and lower CPU utilization that

improves the ability of the instance to handle traffic that is larger in packet volume.

Web Application Delivery & Domain Name Resolution

Services that are available in AWS edge locations, like CloudFront and Route 53, allows

RADAR to significantly increase its ability to optimize latency and throughput to end-users,

absorb DDoS attacks, and isolate faults while minimizing availability impact.

CloudFront only accepts well-formed connections to prevent many common DDoS attacks like

SYN floods and UDP reflection attacks. DDoS attacks are geographically isolated (close to the

source) which prevents the traffic from affecting other locations.

Route 53 is a highly available and scalable domain name system (DNS) service that is used to

direct traffic to RADAR. It includes advanced features like traffic flow, latency-based routing,

Geo DNS, health checks, and monitoring. Even if the DNS service is targeted by a DDoS attack,

with shuffle sharding, each name server in a RADAR record set corresponds to a unique set of

edge locations and Internet paths. This provides greater fault tolerance and minimizes overlap

RADAR DDoS Risk Mitigation v1.2-2020-06-25-AJM 2

CONFIDENTIAL

between customers. Anycast striping is also used so that each DNS request is served by the

most optimal location thus spreading load and reducing DNS latency.

Finally, Route 53 can detect anomalies in the source and volume of DNS queries and prioritize

requests from users that are known to be reliable.

Monitoring

We monitor our applications running on AWS using a variety of tools including CloudWatch.

CloudWatch allows us to collect and track metrics, collect and monitor log files, set alarms, and

automatically react to changes in our AWS resources. These metrics include:

**GroupMaxSize** Maximum size of the Auto Scaling group

**Requests** Number of HTTPS requests

**TotalErrorRate** Percentage of requests with HTTP status code 4xx or 5xx

**CPUUtilization** Percentage of allocated EC2 compute units currently in use

**HTTPCode\_ELB\_4xx**

**HTTPCode\_ELB\_5xx**

The number of HTTP 4xx or 5xx error codes generated by the load balancer

**NetworkIn** The number of bytes received on all network interfaces by the instance

**SurgeQueueLength** The number of requests queued by the load balancer, awaiting a back-end

instance to accept connections and process the request

**UnHealthyHostCount** The number of unhealthy instances in each Availability Zone

**RequestCount** The number of completed requests that were received and routed to registered

instances

**Latency** The time elapsed, in seconds, after the request leaves the load balancer until a

response is received

**BackendConnectionErrors** The number of connections that were not successful

**SpilloverCount** The number of requests that were rejected because the queue was full

**HealthCheckStatus** The status of the health check endpoint

Another tool we can use to gain visibility into traffic targeting RADAR is VPC Flow Logs.

Attack Surface Reduction

In application development, we always look to limit the opportunities that an attacker may have

to target RADAR by limiting the extent to which the application is exposed to the Internet. We

use AWS security groups and network ACLs to limit exposure and control access.

RADAR DDoS Risk Mitigation v1.2-2020-06-25-AJM 3

CONFIDENTIAL

Policy Compliance

Compliance Measurement

The OpSec team will verify compliance to this policy through various methods, including but not

limited to, periodic walkthroughs, business tool reports, internal and external audits, and

feedback to the policy owner.

Training

Based on role and applicability, employees must complete policy-related training every other

year. This may include web-based training courses or instructor-led training programs.

Exceptions

Any exception to the policy must be approved by the OpSec team in advance.

Non-Compliance

An employee found to have violated this policy may be subject to disciplinary action, up to and

including termination of employment in addition to any civil and criminal liability. See Formal

Sanctions Policy for more details.

Maintenance

This policy will be reviewed by the Security Officer and the Privacy Officer at least once a year

or as deemed appropriate based on changes in technology, business or regulatory

requirements.

RADAR DDoS Risk Mitigation v1.2-2020-06-25-AJM 4