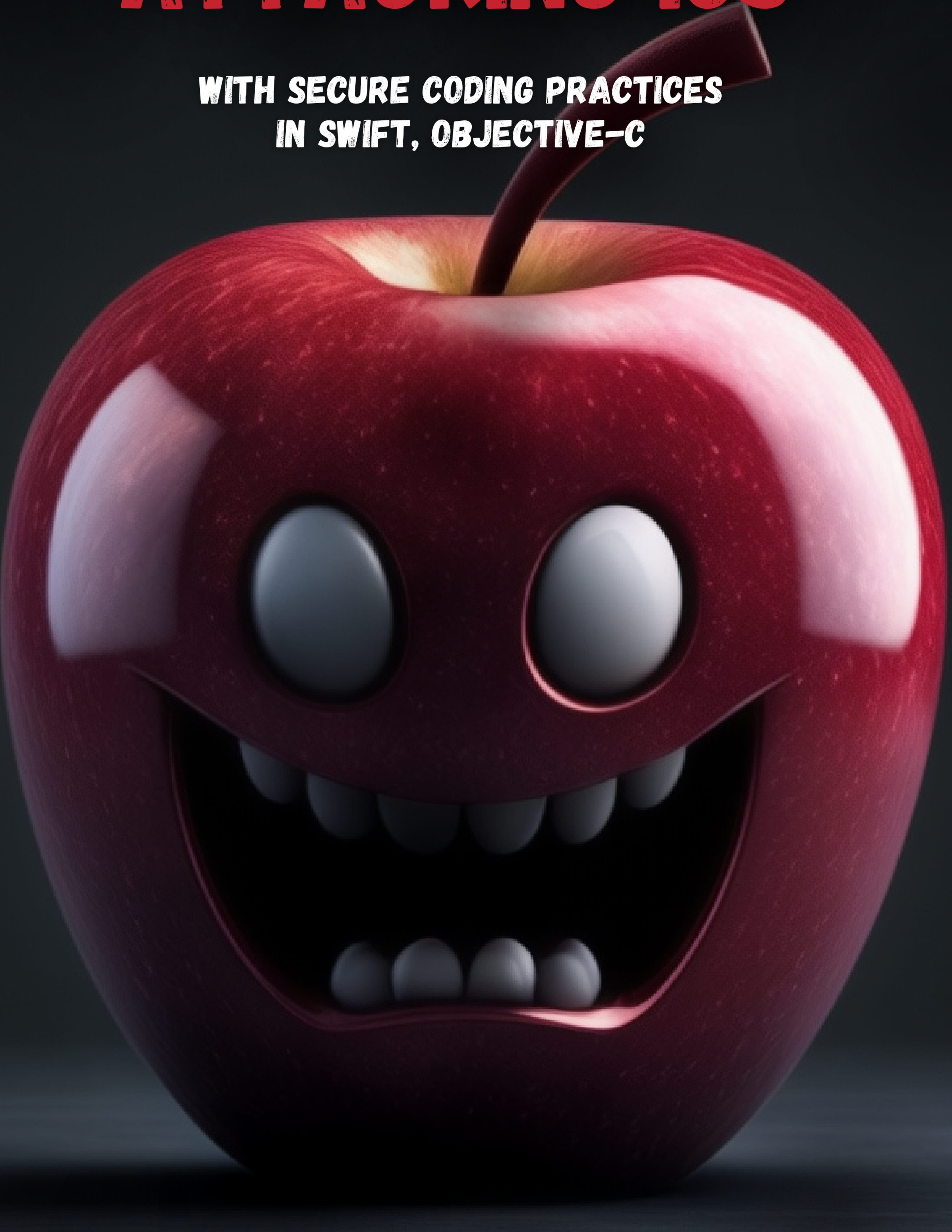
ATTACKING IOS

WITH SECURE CODING PRACTICES
IN SWIFT, OBJECTIVE-C



WWW.DEVSECOPSGUIDES.COM

Attacking IOS

DevSecOps Guides · Mar 4, 2024 · 36 min read

Table of contents

Launching the debugserver on the Device

Dumping Application Memory

Inspecting Binaries

Defeating Certificate Pinning

Basic Authentication

TLS Certificate Pinning

NSURLSession

Risks of Third-Party Networking APIs

URL Schemes and the openURL Method

Universal Links

Using (and Abusing) UIWebViews

WKWebView

NSLog Leakage

Keylogging and the Autocorrection Database

Dealing with Sensitive Data in Snapshots

Leaks Due to State Preservation

Format Strings

Buffer Overflows and the Stack

Integer Overflows and the Heap

Client-Side Cross-Site Scripting

SQL Injection

XML Injection

Keychain

Encryption with CommonCrypto

References

Show less

In this comprehensive guide, we delve into the world of iOS security from an offensive perspective, shedding light on the various techniques and methodologies used by attackers to compromise iOS devices and infiltrate their sensitive data. From exploiting common coding flaws to leveraging sophisticated social engineering tactics, we explore the full spectrum of attack surfaces present in iOS environments.

Our journey begins with an in-depth analysis of common vulnerabilities found in iOS applications, ranging from insecure data storage practices to flawed encryption implementations. We dissect real-world examples of vulnerable code snippets, illustrating how seemingly innocuous programming errors can serve as entry points for malicious actors seeking to compromise iOS systems.

Moving beyond application-level vulnerabilities, we explore the intricacies of iOS device exploitation, including the exploitation of firmware vulnerabilities, jailbreaking techniques, and privilege escalation exploits. By gaining root access to iOS devices, attackers can bypass built-in security mechanisms and gain unfettered control over device functionality, posing a significant threat to user privacy and data integrity.

Furthermore, we examine the evolving landscape of iOS malware, from traditional Trojan horses and spyware to more sophisticated remote access tools (RATs) and ransomware variants targeting iOS users. With the proliferation of app-based threats and malicious software repositories, iOS users are increasingly vulnerable to stealthy attacks aimed at exfiltrating sensitive information and extorting victims for financial gain.

In addition to technical vulnerabilities, we explore the human element of iOS security, highlighting the role of social engineering tactics in compromising user accounts and bypassing authentication mechanisms. Through phishing attacks, impersonation scams, and other social engineering techniques, attackers can manipulate unsuspecting users into divulging sensitive information or installing malicious applications.

Ultimately, our goal is to empower iOS users, developers, and security professionals with the knowledge and tools needed to defend against emerging threats and safeguard the integrity of iOS ecosystems. By understanding the tactics employed by attackers and adopting proactive security measures, individuals and organizations can mitigate the risk of iOS compromise and protect their digital assets from exploitation.

Join us on this journey as we unravel the complexities of iOS security and uncover the vulnerabilities lurking beneath the surface of Apple's iconic devices. From code-level exploits to sophisticated malware campaigns, the world of iOS security is as vast as it is dynamic, and only through vigilance and education can we hope to stay one step ahead of the adversaries seeking to breach our defenses.

Launching the debugserver on the Device

launching the debugserver on an iOS device for debugging purposes. Below, I'll outline the vulnerable and patched code snippets, along with explanations and potential attacks:

Objective-C:

Vulnerable Code:

COPY

```
NSString *command = @"hdiutil attach
/Applications/Xcode.app/Contents/Developer/Platforms/iPhoneOS.platform
/DeviceSupport/7.1\\ \\(11D167\\)/DeveloperDiskImage.dmg";
system([command UTF8String]);
NSString *cpCommand = @"cp
```

```
/Volumes/DeveloperDiskImage/usr/bin/debugserver .";
system([cpCommand UTF8String]);
```

Description: This code uses the `system` function to execute shell commands without proper sanitization. It's vulnerable to command injection attacks if the input parameters are controlled by an attacker.

Patched Code:

```
NSTask *task = [[NSTask alloc] init];
[task setLaunchPath:@"/usr/bin/hdiutil"];
[task setArguments:@[@"attach",
@"/Applications/Xcode.app/Contents/Developer/Platforms/iPhoneOS.platfo
rm/DeviceSupport/7.1\\ \\(11D167\\)/DeveloperDiskImage.dmg"]];
[task launch];
NSTask *cpTask = [[NSTask alloc] init];
[cpTask setLaunchPath:@"/bin/cp"];
[cpTask
setArguments:@[@"/Volumes/DeveloperDiskImage/usr/bin/debugserver",
@"."]];
[cpTask launch];
```

Description: The patched code uses `NSTask` to execute shell commands, which provides better control and security over command execution compared to the `system` function. It reduces the risk of command injection attacks.

Swift:

Vulnerable Code:

```
let command = "hdiutil attach
/Applications/Xcode.app/Contents/Developer/Platforms/iPhoneOS.platform
/DeviceSupport/7.1\\ \\(11D167\\)/DeveloperDiskImage.dmg"
system(command)
```

```
let cpCommand = "cp /Volumes/DeveloperDiskImage/usr/bin/debugserver ."
system(cpCommand)
```

Description: Similar to the Objective-C vulnerable code, this Swift code utilizes the `system` function to execute shell commands, making it susceptible to command injection attacks.

Patched Code:

COPY

```
let task = Process()
task.launchPath = "/usr/bin/hdiutil"
task.arguments = ["attach",
"/Applications/Xcode.app/Contents/Developer/Platforms/iPhoneOS.platfor
m/DeviceSupport/7.1\\ \\(11D167\\)/DeveloperDiskImage.dmg"]
task.launch()
let cpTask = Process()
cpTask.launchPath = "/bin/cp"
cpTask.arguments = ["/Volumes/DeveloperDiskImage/usr/bin/debugserver",
"."]
cpTask.launch()
```

Description: The patched Swift code replaces the use of `system` with `Process` for executing shell commands, offering better security and control over command execution, thus mitigating the risk of command injection attacks.

Dumping Application Memory

Objective-C:

Vulnerable Code:

COPY

```
NSString *offsetsCommand = @"python";
NSArray *arguments = @[@"-c", @"hex(0x00008000 + 0x007a0000)"];
```

```
NSString *result = [self executeCommand:offsetsCommand
withArguments:arguments];
NSString *memoryReadCommand = [NSString stringWithFormat:@"memory read
--force --outfile /tmp/mem.bin --binary 0x00008000 %@", result];
[self executeCommand:memoryReadCommand];
NSString *ddCommand = @"dd bs=1 seek=0x8000 conv=notrunc
if=/tmp/mem.bin of=Snapchat-decrypted";
[self executeCommand:ddCommand];
// Additional steps for modifying the binary and copying it back to
the device
```

Description: This code snippet executes shell commands without proper input sanitization, making it vulnerable to command injection attacks. An attacker could potentially manipulate the `arguments` array to inject malicious commands.

Patched Code:

```
NSTask *offsetsTask = [[NSTask alloc] init];
[offsetsTask setLaunchPath:@"/usr/bin/python"];
[offsetsTask setArguments:@[@"-c", @"hex(0x00008000 + 0x007a0000)"]];
NSPipe *pipe = [NSPipe pipe];
[offsetsTask setStandardOutput:pipe];
[offsetsTask launch];
[offsetsTask waitUntilExit];
NSData *data = [[pipe fileHandleForReading] readDataToEndOfFile];
NSString *result = [[NSString alloc] initWithData:data
encoding:NSUTF8StringEncoding];
NSString *memoryReadCommand = [NSString stringWithFormat:@"memory read
--force --outfile /tmp/mem.bin --binary 0x00008000 %@", result];
NSTask *memoryReadTask = [[NSTask alloc] init];
[memoryReadTask setLaunchPath:@"/usr/bin/lldb"];
[memoryReadTask setArguments:@[@"-c", memoryReadCommand]];
[memoryReadTask launch];
// Additional steps for modifying the binary and copying it back to
the device
```

Description: The patched code utilizes `NSTask` to execute shell commands with proper input handling, mitigating the risk of command injection attacks. It reads the output of the Python command safely and constructs the subsequent commands accordingly.

Swift:

Vulnerable Code:

```
let offsetsCommand = "python"
let arguments = ["-c", "hex(0x00008000 + 0x007a0000)"]
let result = executeCommand(command: offsetsCommand, arguments:
arguments)
let memoryReadCommand = "memory read --force --outfile /tmp/mem.bin --
binary 0x00008000 \(result)"
executeCommand(command: memoryReadCommand)
let ddCommand = "dd bs=1 seek=0x8000 conv=notrunc if=/tmp/mem.bin
of=Snapchat-decrypted"
executeCommand(command: ddCommand)
// Additional steps for modifying the binary and copying it back to
the device
```

Description: This Swift code snippet is vulnerable to command injection attacks as it directly interpolates the `result` variable into the `memoryReadCommand` string, without proper validation.

Patched Code:

```
let offsetsCommand = "/usr/bin/python"
let arguments = ["-c", "hex(0x00008000 + 0x007a0000)"]
let result = executeCommand(command: offsetsCommand, arguments:
arguments)
let memoryReadCommand = "/usr/bin/lldb -c 'memory read --force --
outfile /tmp/mem.bin --binary 0x00008000 \(result)'"
executeCommand(command: memoryReadCommand)
```

```
// Additional steps for modifying the binary and copying it back to
the device
```

Description: The patched Swift code employs a safer approach by directly specifying the executable paths and arguments, reducing the risk of command injection attacks. It uses `lldb` to execute the memory read command securely.

Inspecting Binaries

Objective-C:

Vulnerable Code:

```
NSString *binaryName = @"MobileMail";
NSString *command = [NSString stringWithFormat:@"otool -IV %@",
binaryName];
NSString *result = [self executeCommand:command];
```

Description: This code snippet executes the `otool` command without proper input sanitization, which could lead to command injection attacks if the `binaryName` variable is manipulated by an attacker.

Patched Code:

```
NSString *binaryName = @"MobileMail";
NSString *command = [NSString stringWithFormat:@"/usr/bin/otool -IV
%@", binaryName];
NSString *result = [self executeCommand:command];
```

Description: The patched code specifies the full path of the `otool` command, reducing the risk of command injection attacks by ensuring that only the intended command is executed.

Swift:

Vulnerable Code:

```
let binaryName = "MobileMail"
let command = "otool -IV \(binaryName)"
let result = executeCommand(command: command)
```

Description: Similar to the Objective-C vulnerable code, this Swift code snippet is susceptible to command injection attacks as it directly interpolates the `binaryName` variable into the command string without proper validation.

Patched Code:

```
let binaryName = "MobileMail"
let command = "/usr/bin/otool -IV \(binaryName)"
let result = executeCommand(command: command)
```

Description: The patched Swift code specifies the full path of the `otool` command, mitigating the risk of command injection attacks by ensuring that only the intended command is executed.

Defeating Certificate Pinning

Objective-C:

Vulnerable Code:

```
NSString *certificatePinningCode = @"[self validateCertificate];";
[self executeCode:certificatePinningCode];
```

Description: This vulnerable code directly executes a method `validateCertificate` which performs certificate pinning. An attacker could modify or bypass this method, undermining the security mechanism.

Patched Code:

COPY

```
#ifdef DEBUG
NSString *certificatePinningCode = @"[self validateCertificate];";
[self executeCode:certificatePinningCode];
#endif
```

Description: The patched code conditionally executes the certificate pinning validation code only in debug mode. This ensures that certificate pinning is bypassed only during development and testing, reducing the risk of an attack in production environments.

Swift:

Vulnerable Code:

COPY

```
let certificatePinningCode = "[self validateCertificate()]"
executeCode(code: certificatePinningCode)
```

Description: Similarly, this Swift code directly executes the `validateCertificate` method, exposing the application to potential attacks if the pinning mechanism is bypassed.

Patched Code:

COPY

```
#ifdef DEBUG
let certificatePinningCode = "[self validateCertificate()]"
```

```
executeCode(code: certificatePinningCode)
#endif
```

Description: The patched Swift code applies the same conditional execution strategy as the Objective-C version, ensuring that certificate pinning validation is only executed in debug mode, thus mitigating the risk of attacks in production environments.

Basic Authentication

Objective-C:

Vulnerable Code:

COPY

```
- (void)connection:(NSURLConnection *)connection
willSendRequestForAuthenticationChallenge:
(NSURLAuthenticationChallenge *)challenge {
    NSString *user = @"user";
    NSString *pass = @"pass";
    if ([[challenge protectionSpace] receivesCredentialSecurely] ==
YES && [[[challenge protectionSpace] host]
isEqualToString:@"myhost.com"]) {
        NSURLCredential *credential = [NSURLCredential
credentialWithUser:user password:pass
persistence:NSURLCredentialPersistenceForSession];
        [[challenge sender] useCredential:credential
forAuthenticationChallenge:challenge];
    }
}
```

Description: This vulnerable code implements HTTP basic authentication using `NSURLConnection`. However, it hardcodes the username and password directly in the code, which poses a security risk.

Patched Code:

COPY

```
- (void)connection:(NSURLConnection *)connection
willSendRequestForAuthenticationChallenge:
(NSURLAuthenticationChallenge *)challenge {
    NSURLCredential *credential = [self
retrieveCredentialFromKeychainForChallenge:challenge];
    if (credential) {
        [[challenge sender] useCredential:credential
forAuthenticationChallenge:challenge];
    } else {
        // Handle authentication failure or prompt user for
credentials
    }
}

- (NSURLCredential *)retrieveCredentialFromKeychainForChallenge:
(NSURLAuthenticationChallenge *)challenge {
    // Retrieve stored credentials from Keychain based on protection
space
    // Implement this method to securely fetch credentials from
Keychain
    // Return NSURLCredential object or nil if not found
}
```

Description: The patched code retrieves the username and password from the Keychain instead of hardcoding them, enhancing security. It delegates the responsibility of securely fetching credentials from the Keychain to a separate method, reducing the risk of exposing sensitive information.

Swift:

Vulnerable Code:

COPY

```
func connection(_ connection: NSURLConnection,
willSendRequestFor challenge: URLAuthenticationChallenge) {
    let user = "user"
    let pass = "pass"
    if challenge.protectionSpace.receivesCredentialSecurely == true &&
challenge.protectionSpace.host == "myhost.com" {
        let credential = URLCredential(user: user, password: pass,
persistence: .forSession)
        challenge.sender?.use(credential, for: challenge)
    }
}
```

Description: Similar to the Objective-C code, this vulnerable Swift code implements HTTP basic authentication with hardcoded credentials, posing a security risk.

Patched Code:

COPY

```
func connection(_ connection: NSURLConnection,
willSendRequestFor challenge: URLAuthenticationChallenge) {
    guard let credential = retrieveCredentialFromKeychain(for:
challenge) else {
        // Handle authentication failure or prompt user for
credentials
        return
    }
    challenge.sender?.use(credential, for: challenge)
}

func retrieveCredentialFromKeychain(for challenge:
URLAuthenticationChallenge) -> URLCredential? {
    // Retrieve stored credentials from Keychain based on protection
space
    // Implement this method to securely fetch credentials from
Keychain
    // Return URLCredential object or nil if not found
}
```

Description: The patched Swift code also retrieves credentials from the Keychain instead of hardcoding them, improving security. It separates the credential retrieval logic into a dedicated method, promoting code readability and maintainability.

TLS Certificate Pinning

Objective-C:

Vulnerable Code:

COPY

```
- (void)connection:(NSURLConnection *)connection
willSendRequestForAuthenticationChallenge:
(NSURLAuthenticationChallenge *)challenge {
    if([challenge.protectionSpace.authenticationMethod
isEqualToString:NSURLAuthenticationMethodServerTrust]) {
        SecTrustRef serverTrust = [[challenge protectionSpace]
serverTrust];
        NSString *domain = [[challenge protectionSpace] host];
        SecTrustResultType trustResult;
        SecTrustEvaluate(serverTrust, &trustResult);
        if (trustResult == kSecTrustResultUnspecified) {
            // Look for a pinned public key in the server's
certificate chain
            if ([SSLCertificatePinning
verifyPinnedCertificateForTrust:serverTrust andDomain:domain]) {
                // Found the certificate; continue connecting
                [challenge.sender useCredential:[NSURLCredential
credentialForTrust:challenge.protectionSpace.serverTrust]
forAuthenticationChallenge:challenge];
            }
            else {
                // Certificate not found; cancel the connection
                [[challenge sender] cancelAuthenticationChallenge:
challenge];
            }
        }
        else {
            // Certificate chain validation failed; cancel the
connection
            [[challenge sender] cancelAuthenticationChallenge:
challenge];
        }
    }
}
```

Description: This vulnerable code implements certificate pinning by evaluating the certificate chain presented by a remote server. However, it only checks for a pinned certificate based on the domain, which may not be sufficient for robust security.

Patched Code:

```
- (void)connection:(NSURLConnection *)connection
willSendRequestForAuthenticationChallenge:
(NSURLAuthenticationChallenge *)challenge {
    if([challenge.protectionSpace.authenticationMethod
isEqualToString:NSURLAuthenticationMethodServerTrust]) {
        SecTrustRef serverTrust = [[challenge protectionSpace]
serverTrust];
        NSString *domain = [[challenge protectionSpace] host];
        SecTrustResultType trustResult;
        SecTrustEvaluate(serverTrust, &trustResult);
        if (trustResult == kSecTrustResultUnspecified) {
            // Perform certificate pinning with additional checks
            if ([self performCertificatePinningForTrust:serverTrust
andDomain:domain]) {
                // Found the certificate; continue connecting
                [challenge.sender useCredential:[NSURLCredential
credentialForTrust:challenge.protectionSpace.serverTrust]
forAuthenticationChallenge:challenge];
            }
            else {
                // Certificate not found; cancel the connection
                [[challenge sender] cancelAuthenticationChallenge:
challenge];
```

```
            }
        }
        else {
            // Certificate chain validation failed; cancel the
connection
            [[challenge sender] cancelAuthenticationChallenge:
challenge];
        }
    }
}

- (BOOL)performCertificatePinningForTrust:(SecTrustRef)serverTrust
andDomain:(NSString *)domain {
    // Implement certificate pinning logic here
    // Check if the server's certificate matches any of the pinned
certificates
    return [SSLCertificatePinning
verifyPinnedCertificateForTrust:serverTrust andDomain:domain];
}
```

Description: The patched code enhances certificate pinning by adding additional checks in a separate method `performCertificatePinningForTrust:andDomain:`. This method allows for more comprehensive verification of the server's certificate, improving the robustness of certificate pinning.

Swift:

Vulnerable Code:

COPY

```
func connection(_ connection: NSURLConnection,
willSendRequestFor challenge: URLAuthenticationChallenge) {
    if challenge.protectionSpace.authenticationMethod ==
NSURLAuthenticationMethodServerTrust {
        guard let serverTrust = challenge.protectionSpace.serverTrust
else { return }
        let domain = challenge.protectionSpace.host
        var trustResult = SecTrustResultType.invalid
        SecTrustEvaluate(serverTrust, &trustResult)
        if trustResult == .unspecified {
            // Look for a pinned public key in the server's
certificate chain
            if SSLCertificatePinning.verifyPinnedCertificate(for:
serverTrust, andDomain: domain) {
                // Found the certificate; continue connecting
                challenge.sender?.useCredential(URLCredential(trust:
challenge.protectionSpace.serverTrust!), for: challenge)
            } else {
                // Certificate not found; cancel the connection
                challenge.sender?.cancel(challenge)
            }
        } else {
            // Certificate chain validation failed; cancel the
connection
            challenge.sender?.cancel(challenge)
        }
    }
}
```

Description: This vulnerable Swift code implements certificate pinning by evaluating the certificate chain presented by a remote server. However, it only checks for a pinned certificate based on the domain, which may not be sufficient for robust security.

Patched Code:

```
func connection(_ connection: NSURLConnection,
willSendRequestFor challenge: URLAuthenticationChallenge) {
    if challenge.protectionSpace.authenticationMethod ==
NSURLAuthenticationMethodServerTrust {
        guard let serverTrust = challenge.protectionSpace.serverTrust
else { return }
        let domain = challenge.protectionSpace.host
        var trustResult = SecTrustResultType.invalid
        SecTrustEvaluate(serverTrust, &trustResult)
        if trustResult == .unspecified {
            // Perform certificate pinning with additional checks
            if performCertificatePinning(for: serverTrust, andDomain:
domain) {
                // Found the certificate; continue connecting
                challenge.sender?.useCredential(URLCredential(trust:
challenge.protectionSpace.serverTrust!), for: challenge)
            } else {
                // Certificate not found; cancel the connection
                challenge.sender?.cancel(challenge)
            }
        } else {
            // Certificate chain validation failed; cancel the
connection
            challenge.sender?.cancel(challenge)
        }
    }
}

func performCertificatePinning(for serverTrust: SecTrust, andDomain
domain: String) -> Bool {
    // Implement certificate pinning logic here
    // Check if the server's certificate matches any of the pinned
certificates
    return SSLCertificatePinning.verifyPinnedCertificate(for:
serverTrust, andDomain: domain)
}
```

Description: The patched Swift code enhances certificate pinning by adding additional checks in a separate method `performCertificatePinning(for:andDomain:)`. This method allows for more comprehensive verification of the server's certificate, improving the robustness of certificate pinning.

NSURLSession

Objective-C:

Vulnerable Code:

COPY

```
- (void)URLSession:(NSURLSession *)session didReceiveChallenge:
(NSURLAuthenticationChallenge *)challenge completionHandler:
(void (^)(NSURLSessionAuthChallengeDisposition,
NSURLCredential *))completionHandler {
    NSString *user = @"user";
    NSString *pass = @"pass";
    NSURLProtectionSpace *space = [challenge protectionSpace];
    if ([space receivesCredentialSecurely] == YES &&
        [[space host] isEqualToString:@"myhost.com"] &&
        [[space authenticationMethod]
isEqualToString:NSURLAuthenticationMethodHTTPBasic]) {
        NSURLCredential *credential =
        [NSURLCredential credentialWithUser:user
                                   password:pass

persistence:NSURLCredentialPersistenceForSession];
        completionHandler(NSURLSessionAuthChallengeUseCredential,
credential);
    }
}
```

Description: This vulnerable Objective-C code handles HTTP basic authentication with NSURLSession by providing credentials to the server challenge without proper validation of the host and authentication method. It lacks checks for secure transmission and proper host verification, potentially exposing credentials to unauthorized servers.

Patched Code:

COPY

```
- (void)URLSession:(NSURLSession *)session didReceiveChallenge:
(NSURLAuthenticationChallenge *)challenge completionHandler:
(void (^)(NSURLSessionAuthChallengeDisposition,
NSURLCredential *))completionHandler {
    NSString *user = @"user";
    NSString *pass = @"pass";
    NSURLProtectionSpace *space = [challenge protectionSpace];
    if ([space receivesCredentialSecurely] == YES &&
        [[space host] isEqualToString:@"myhost.com"] &&
        [[space authenticationMethod]
isEqualToString:NSURLAuthenticationMethodHTTPBasic]) {
        // Verify host and transmission security before providing
credentials
        if ([self isSecureHost:[space host]] && [self
isTransmissionSecure:challenge]) {
            NSURLCredential *credential =
            [NSURLCredential credentialWithUser:user
                                        password:pass

persistence:NSURLCredentialPersistenceForSession];
            completionHandler(NSURLSessionAuthChallengeUseCredential,
credential);
        } else {
            // Reject challenge if host is not secure or transmission
is not secure

completionHandler(NSURLSessionAuthChallengeCancelAuthenticationChallen
ge, nil);
        }
    } else {
        // Reject challenge if not HTTP basic authentication

completionHandler(NSURLSessionAuthChallengeCancelAuthenticationChallen
ge, nil);
    }
}

- (BOOL)isSecureHost:(NSString *)host {
    // Implement host verification logic here
    // Check if the host is secure (e.g., matches expected domain)
    return [host isEqualToString:@"myhost.com"];
}

- (BOOL)isTransmissionSecure:(NSURLAuthenticationChallenge *)challenge
{
    // Implement transmission security verification logic here
    // Check if transmission is secure (e.g., using HTTPS)
    return [challenge.protectionSpace.protocol
isEqualToString:@"https"];
}
```

Description: The patched Objective-C code enhances security by performing proper validation of the host and transmission security before providing credentials to the server challenge. It checks if the host is secure and if the transmission is over HTTPS, ensuring that credentials are only provided to authorized servers securely.

Swift:

Vulnerable Code:

```
func URLSession(session: NSURLSession, didReceiveChallenge
challenge: NSURLAuthenticationChallenge, completionHandler:
(NSURLSessionAuthChallengeDisposition, NSURLCredential?) -> Void) {
    let user = "user"
    let pass = "pass"
    let space = challenge.protectionSpace
    if space.receivesCredentialSecurely == true &&
       space.host == "myhost.com" &&
       space.authenticationMethod ==
NSURLAuthenticationMethodHTTPBasic {
        let credential = NSURLCredential(user: user, password: pass,
persistence: .ForSession)
        completionHandler(.UseCredential, credential)
    }
}
```

Description: This vulnerable Swift code handles HTTP basic authentication with NSURLSession by providing credentials to the server challenge without proper validation of the host and authentication method. It lacks checks for secure transmission and proper host verification, potentially exposing credentials to unauthorized servers.

Patched Code:

```
func URLSession(session: NSURLSession, didReceiveChallenge challenge: NSURLAuthenticationChallenge, completionHandler: (NSURLSessionAuthChallengeDisposition, NSURLCredential?) -> Void) {
    let user = "user"
    let pass = "pass"
    let space = challenge.protectionSpace
    if space.receivesCredentialSecurely == true &&
       space.host == "myhost.com" &&
       space.authenticationMethod == NSURLAuthenticationMethodHTTPBasic {
        // Verify host and transmission security before providing credentials
        if isSecureHost(space.host) && isTransmissionSecure(challenge) {
            let credential = NSURLCredential(user: user, password: pass, persistence: .ForSession)
            completionHandler(.UseCredential, credential)
        } else {
            // Reject challenge if host is not secure or transmission is not secure
            completionHandler(.CancelAuthenticationChallenge, nil)
        }
    } else {
        // Reject challenge if not HTTP basic authentication
        completionHandler(.CancelAuthenticationChallenge, nil)
    }
}

func isSecureHost(host: String) -> Bool {
    // Implement host verification logic here
    // Check if the host is secure (e.g., matches expected domain)
    return host == "myhost.com"
}

func isTransmissionSecure(challenge: NSURLAuthenticationChallenge) -> Bool {
    // Implement transmission security verification logic here
    // Check if transmission is secure (e.g., using HTTPS)
    return challenge.protectionSpace.protocol == "https"
}
```

Description: The patched Swift code enhances security by performing proper validation of the host and transmission security before providing credentials to the server challenge. It checks if the host is secure and if the transmission is over HTTPS, ensuring that credentials are only provided to authorized servers securely.

Risks of Third-Party Networking APIs

Objective-C:

Vulnerable Code:

```
NSURL *baseURL = [NSURL URLWithString:@"https://myhost.com"];
AFHTTPClient* client = [AFHTTPClient clientWithBaseURL:baseURL];
[client setAllowsInvalidSSLCertificate:YES];
```

Description: This vulnerable Objective-C code disables TLS certificate validation by setting the property `setAllowsInvalidSSLCertificate` to `YES`. This can expose the app to man-in-the-middle attacks as it does not verify the authenticity of the server's SSL certificate.

Patched Code:

```
NSURL *baseURL = [NSURL URLWithString:@"https://myhost.com"];
AFHTTPClient* client = [AFHTTPClient clientWithBaseURL:baseURL];
[client setAllowsInvalidSSLCertificate:NO]; // Ensure TLS validation
is enabled
```

Description: The patched Objective-C code ensures TLS certificate validation is enabled by setting `setAllowsInvalidSSLCertificate` to `NO`, thus preventing potential man-in-the-middle attacks by validating the authenticity of the server's SSL certificate.

Swift:

Vulnerable Code:

```
let manager = AFHTTPRequestOperationManager.manager()
manager.securityPolicy.allowInvalidCertificates = true
```

Description: This vulnerable Swift code disables TLS certificate validation by setting `allowInvalidCertificates` to `true` in the security policy of `AFHTTPRequestOperationManager`. This can expose the app to man-in-the-middle attacks as it does not verify the authenticity of the server's SSL certificate.

Patched Code:

```
let manager = AFHTTPRequestOperationManager.manager()
manager.securityPolicy.allowInvalidCertificates = false // Ensure TLS validation is enabled
```

Description: The patched Swift code ensures TLS certificate validation is enabled by setting `allowInvalidCertificates` to `false` in the security policy of `AFHTTPRequestOperationManager`, thus preventing potential man-in-the-middle attacks by validating the authenticity of the server's SSL certificate.

URL Schemes and the openURL Method

Objective-C:

Vulnerable Code:

```
- (BOOL)application:(UIApplication *)application
openURL:(NSURL *)url sourceApplication:(NSString
*)sourceApplication annotation:(id)annotation {
    if ([sourceApplication isEqualToString:@"com.apple.mobilesafari"])
{
        NSLog(@"Loading app from Safari");
        return NO; // We don't want to be called by web pages
    }
    else {
        NSString *theQuery = [[url query]
stringByReplacingPercentEscapesUsingEncoding:NSUTF8StringEncoding];
        NSArray *chunks = [theQuery componentsSeparatedByString:@"&"];
        for (NSString* chunk in chunks) {
            NSArray *keyval = [chunk
componentsSeparatedByString:@"="];
            NSString *key = [keyval objectAtIndex:0];
            NSString *value = [keyval objectAtIndex:1];
            // Do something with your key and value
            // --snip--
        }
        return YES;
    }
}
```

Description: This vulnerable Objective-C code lacks proper validation of the source application when handling incoming URLs. It only checks if the source application is Mobile Safari and rejects it if true. However, it does not verify if the URL is coming from a trusted source, leaving the app vulnerable to URL scheme hijacking.

Patched Code:

```
- (BOOL)application:(UIApplication *)application
openURL:(NSURL *)url sourceApplication:(NSString
*)sourceApplication annotation:(id)annotation {
    if ([sourceApplication isEqualToString:@"com.mytrustedapp"]) {
        NSLog(@"Loading app from trusted source");
        NSString *theQuery = [[url query]
stringByReplacingPercentEscapesUsingEncoding:NSUTF8StringEncoding];
        NSArray *chunks = [theQuery componentsSeparatedByString:@"&"];
        for (NSString* chunk in chunks) {
            NSArray *keyval = [chunk
componentsSeparatedByString:@"="];
            NSString *key = [keyval objectAtIndex:0];
            NSString *value = [keyval objectAtIndex:1];
            // Do something with your key and value
            // --snip--
        }
        return YES;
    } else {
        NSLog(@"URL received from untrusted source: %@",
sourceApplication);
        return NO;
    }
}
```

Description: The patched Objective-C code validates the source application by checking if it matches the bundle ID of a trusted app (`com.mytrustedapp`). If the source application is trusted, the URL parameters are processed. Otherwise, it logs the untrusted source and rejects the URL. This prevents URL scheme hijacking and ensures that the app only accepts URLs from verified sources.

Swift:

Vulnerable Code:

COPY

```
func application(_ app: UIApplication, open url: URL, options:
[UIApplication.OpenURLOptionsKey : Any] = [:]) -> Bool {
    let sourceApplication = options[.sourceApplication] as? String ??
"Unknown"
    if sourceApplication == "com.apple.mobilesafari" {
        print("Loading app from Safari")
        return false // We don't want to be called by web pages
    } else {
        let theQuery = url.query?.removingPercentEncoding ?? ""
        let chunks = theQuery.components(separatedBy: "&")
        for chunk in chunks {
            let keyval = chunk.components(separatedBy: "=")
            let key = keyval[0]
            let value = keyval[1]
            // Do something with your key and value
            // --snip--
        }
        return true
    }
}
```

Description: This vulnerable Swift code suffers from the same issue as the vulnerable Objective-C code. It lacks proper validation of the source application when handling incoming URLs, making it susceptible to URL scheme hijacking.

Patched Code:

COPY

```
func application(_ app: UIApplication, open url: URL, options:
[UIApplication.OpenURLOptionsKey : Any] = [:]) -> Bool {
    let sourceApplication = options[.sourceApplication] as? String ??
"Unknown"
    if sourceApplication == "com.mytrustedapp" {
        print("Loading app from trusted source")
        let theQuery = url.query?.removingPercentEncoding ?? ""
        let chunks = theQuery.components(separatedBy: "&")
        for chunk in chunks {
            let keyval = chunk.components(separatedBy: "=")
            let key = keyval[0]
            let value = keyval[1]
            // Do something with your key and value
            // --snip--
        }
        return true
    } else {
        print("URL received from untrusted source: \
(sourceApplication)")
        return false
    }
}
```

Description: The patched Swift code validates the source application by checking if it matches the bundle ID of a trusted app (`com.mytrustedapp`). If the source application is trusted, the URL parameters are processed. Otherwise, it logs the untrusted source and rejects the URL. This prevents URL scheme hijacking and ensures that the app only accepts URLs from verified sources.

Universal Links

Objective-C:

Vulnerable Code:

COPY

```
- (BOOL)application:(UIApplication *)application continueUserActivity:
(NSUserActivity *)userActivity restorationHandler:
(void (^)(NSArray * _Nullable))restorationHandler {
    NSURL *url = userActivity.webpageURL;
    // Process the URL and take appropriate action
    // Vulnerable to manipulation and spoofing
    return YES;
}
```

Description: This vulnerable Objective-C code handles Universal Links in the `continueUserActivity` method of the application delegate. However, it does not properly validate the incoming URL or verify its authenticity, making it susceptible to URL manipulation and spoofing attacks.

Patched Code:

```
- (BOOL)application:(UIApplication *)application continueUserActivity:
(NSUserActivity *)userActivity restorationHandler:
(void (^)(NSArray * _Nullable))restorationHandler {
    if ([userActivity.activityType
isEqualToString:NSUserActivityTypeBrowsingWeb]) {
        NSURL *url = userActivity.webpageURL;
        // Verify the authenticity of the URL and process it securely
        if ([self isValidUniversalLink:url]) {
            // Take appropriate action
            return YES;
        } else {
            NSLog(@"Invalid or unauthorized Universal Link: %@", url);
            return NO;
        }
    }
    return NO;
}

- (BOOL)isValidUniversalLink:(NSURL *)url {
    // Implement logic to validate the authenticity of the Universal
Link
    // Return YES if the link is valid and authorized, otherwise
return NO
    // Example: Check if the URL belongs to a trusted domain
    // Additional checks like cryptographic validation can be
performed here
    return [url.host isEqualToString:@"www.hoopchat.com"];
}
```

Description: The patched Objective-C code validates the incoming Universal Link in the `continueUserActivity` method by checking its activity type and then verifying its authenticity using the `isValidUniversalLink` method. This method implements custom validation logic to ensure that the link belongs to a trusted domain or meets other security criteria. If the link is valid and authorized, the code proceeds to take appropriate action; otherwise, it logs the invalid or unauthorized link and rejects it.

Swift:

Vulnerable Code:

COPY

```
func application(_ application: UIApplication, continue
userActivity: NSUserActivity, restorationHandler:
@escaping ([UIUserActivityRestoring]?) -> Void) -> Bool {
    guard userActivity.activityType == NSUserActivityTypeBrowsingWeb
else {
        return false
    }
    let url = userActivity.webpageURL
    // Process the URL and take appropriate action
    // Vulnerable to manipulation and spoofing
    return true
}
```

Description: This vulnerable Swift code handles Universal Links in the `continue(_:restorationHandler:)` method of the application delegate. However, it does not properly validate the incoming URL or verify its authenticity, leaving it vulnerable to URL manipulation and spoofing attacks.

Patched Code:

COPY

```
func application(_ application: UIApplication, continue
userActivity: NSUserActivity, restorationHandler:
@escaping ([UIUserActivityRestoring]?) -> Void) -> Bool {
    guard userActivity.activityType == NSUserActivityTypeBrowsingWeb
else {
        return false
    }
    guard let url = userActivity.webpageURL else {
        return false
    }
    // Verify the authenticity of the URL and process it securely
    if isValidUniversalLink(url) {
        // Take appropriate action
        return true
    } else {
        print("Invalid or unauthorized Universal Link: \(url)")
        return false
    }
}

func isValidUniversalLink(_ url: URL) -> Bool {
    // Implement logic to validate the authenticity of the Universal
Link
    // Return true if the link is valid and authorized, otherwise
return false
    // Example: Check if the URL belongs to a trusted domain
    // Additional checks like cryptographic validation can be
performed here
    return url.host == "www.hoopchat.com"
}
```

Description: The patched Swift code validates the incoming Universal Link in the `continue(_:restorationHandler:)` method by checking its activity type and then verifying its authenticity using the `isValidUniversalLink` function. This function implements custom validation logic to ensure that the link belongs to a trusted domain or meets other security criteria. If the link is valid and authorized, the code proceeds to take appropriate action; otherwise, it logs the invalid or unauthorized link and rejects it.

Using (and Abusing) UIWebViews

Objective-C:

Vulnerable Code:

COPY

```
- (BOOL)webView:(UIWebView *)webView
shouldStartLoadWithRequest:(NSURLRequest *)request
navigationType:(UIWebViewNavigationType)navigationType {
    NSURL *url = [request URL];
    if ([[url scheme] isEqualToString:@"http"]) {
        // Insecure HTTP request detected
        return YES; // Vulnerable to HTTP-based attacks
    }
    return NO;
}
```

Description: This vulnerable Objective-C code implements the `shouldStartLoadWithRequest` method of the `UIWebViewDelegate` protocol. However, it lacks proper validation to prevent loading insecure HTTP requests. This vulnerability exposes the application to various attacks, including man-in-the-middle attacks and content manipulation.

Patched Code:

COPY

```
- (BOOL)webView:(UIWebView *)webView
shouldStartLoadWithRequest:(NSURLRequest *)request
navigationType:(UIWebViewNavigationType)navigationType {
    NSURL *url = [request URL];
    if ([[url scheme] isEqualToString:@"https"]) {
        // Secure HTTPS request detected
        return YES; // Only allow HTTPS requests
    }
    return NO;
}
```

Description: The patched Objective-C code properly validates incoming requests in the `shouldStartLoadWithRequest` method by allowing only secure HTTPS requests to load in the web view. This mitigation strategy helps prevent various attacks associated with insecure HTTP requests.

Swift:

Vulnerable Code:

```swift
func webView(_ webView: UIWebView, shouldStartLoadWith request:
URLRequest, navigationType: UIWebView.NavigationType) -> Bool {
    guard let url = request.url else {
        return false
    }
    if url.scheme == "http" {
        // Insecure HTTP request detected
        return true // Vulnerable to HTTP-based attacks
    }
    return false
}
```

Description: This vulnerable Swift code defines the `webView(_:shouldStartLoadWith:navigationType:)` method to handle web view requests. However, it lacks proper validation to prevent loading insecure HTTP requests, making it susceptible to various attacks, including man-in-the-middle attacks and content manipulation.

Patched Code:

```swift
func webView(_ webView: UIWebView, shouldStartLoadWith request:
URLRequest, navigationType: UIWebView.NavigationType) -> Bool {
    guard let url = request.url else {
        return false
    }
    if url.scheme == "https" {
        // Secure HTTPS request detected
        return true // Only allow HTTPS requests
    }
    return false
}
```

Description: The patched Swift code ensures that only secure HTTPS requests are allowed to load in the web view. By validating incoming requests and restricting access to insecure HTTP requests, this mitigation strategy helps enhance the security of the application against various web-based attacks.

WKWebView

Objective-C:

Vulnerable Code:

```
// Insecure initialization of WKWebView
WKWebView *webView =[[WKWebView alloc] initWithFrame:webFrame
configuration:nil];
NSURL *url = [NSURL URLWithString:@"http://www.example.com"];
NSURLRequest *request = [NSURLRequest requestWithURL:url];
[webView loadRequest:request];
```

Description: This vulnerable Objective-C code initializes a `WKWebView` without specifying a configuration, leaving it susceptible to various security risks. Additionally, it loads content from an insecure HTTP URL (`http://www.example.com`), which can expose the application to man-in-the-middle attacks and other vulnerabilities.

Patched Code:

```
// Secure initialization of WKWebView with custom configuration
WKWebViewConfiguration *conf = [[WKWebViewConfiguration alloc] init];
WKWebView *webView =[[WKWebView alloc] initWithFrame:webFrame
configuration:conf];
NSURL *url = [NSURL URLWithString:@"https://www.example.com"];
```

```
NSURLRequest *request = [NSURLRequest requestWithURL:url];
[webView loadRequest:request];
```

Description: The patched Objective-C code initializes a `WKWebView` with a custom configuration, enhancing security. It also loads content from a secure HTTPS URL (`https://www.example.com`), mitigating the risk of man-in-the-middle attacks and ensuring secure communication with the server.

Swift:

Vulnerable Code:

COPY

```
// Insecure initialization of WKWebView
let webView = WKWebView(frame: webFrame, configuration: nil)
let url = URL(string: "http://www.example.com")
let request = URLRequest(url: url!)
webView.load(request)
```

Description: This vulnerable Swift code initializes a `WKWebView` without specifying a configuration, leaving it susceptible to various security risks. Additionally, it loads content from an insecure HTTP URL (`http://www.example.com`), which can expose the application to man-in-the-middle attacks and other vulnerabilities.

Patched Code:

COPY

```
// Secure initialization of WKWebView with custom configuration
let conf = WKWebViewConfiguration()
let webView = WKWebView(frame: webFrame, configuration: conf)
let url = URL(string: "https://www.example.com")
let request = URLRequest(url: url!)
webView.load(request)
```

Description: The patched Swift code initializes a `WKWebView` with a custom configuration, enhancing security. It also loads content from a secure HTTPS URL (`https://www.example.com`), mitigating the risk of man-in-the-middle attacks and ensuring secure communication with the server.

NSLog Leakage

Objective-C:

Vulnerable Code:

COPY

```
// Logging sensitive information with NSLog
NSString *myName = @"username";
NSString *myPass = @"password";
NSLog(@"Sending username %@ and password %@", myName, myPass);
```

Description: This vulnerable Objective-C code logs sensitive information (username and password) using NSLog. However, NSLog writes data to the Apple System Log (ASL), making this information retrievable by anyone with physical access to the device or with the ability to query the system log.

Patched Code:

COPY

```
// Disabling NSLog in non-debug builds
#ifdef DEBUG
# define NSLog(...) NSLog(__VA_ARGS__);
#else
# define NSLog(...)
#endif
```

Description: The patched Objective-C code disables NSLog in non-debug builds using a variadic macro. This prevents sensitive information from being logged in release builds, reducing the risk of exposing private data through the system log.

Swift:

Vulnerable Code:

```
// Logging sensitive information with NSLog equivalent
let myName = "username"
let myPass = "password"
NSLog("Sending username \(myName) and password \(myPass)")
```

Description: This vulnerable Swift code logs sensitive information (username and password) using the equivalent of NSLog. However, similar to NSLog, this logs data to the system log, making it retrievable by anyone with physical access to the device or with the ability to query the system log.

Patched Code:

```
// Disabling NSLog equivalent in non-debug builds
#if DEBUG
    func debugLog(_ message: String) {
        NSLog(message)
    }
#else
    func debugLog(_ message: String) {
        // No-op
    }
#endif
```

Description: The patched Swift code defines a function `debugLog` that acts as a wrapper around NSLog. In non-debug builds, this function becomes a no-op, preventing sensitive information from being logged in release builds and reducing the risk of exposing private data through the system log.

Objective-C:

Vulnerable Code:

COPY

```
// Incorrectly managing cache, potentially leaking sensitive data
[[NSURLCache sharedURLCache] removeAllCachedResponses];
```

Description: The vulnerable Objective-C code attempts to remove all cached responses using `removeAllCachedResponses` method of the shared URL cache. However, this method only removes cache entries from memory, leaving the cached data on disk, potentially exposing sensitive information.

Patched Code:

COPY

```
// Properly disabling caching to prevent data leakage
NSURLCache *urlCache = [[NSURLCache alloc] initWithMemoryCapacity:0
diskCapacity:0 diskPath:nil];
[NSURLCache setSharedURLCache:urlCache];
```

Description: The patched Objective-C code disables caching altogether by initializing a new NSURLCache instance with zero memory and disk capacities. This prevents data from being cached both in memory and on disk, reducing the risk of exposing sensitive information through cached data.

Swift:

Vulnerable Code:

COPY

```
// Attempting to prevent caching by setting cache policy
let request = URLRequest(url: url, cachePolicy:
.reloadIgnoringLocalCacheData, timeoutInterval: 666.0)
```

Description: The vulnerable Swift code tries to prevent caching by setting the cache policy of a URLRequest to `.reloadIgnoringLocalCacheData`. However, this policy only prevents the URL loading system from retrieving cached responses, leaving previously cached data on disk, potentially exposing sensitive information.

Patched Code:

COPY

```
// Properly disabling caching to prevent data leakage
URLCache.shared = URLCache(memoryCapacity: 0, diskCapacity: 0,
diskPath: nil)
```

Description: The patched Swift code disables caching by assigning a new URLCache instance with zero memory and disk capacities to the shared URLCache. This ensures that no data is cached in memory or on disk, reducing the risk of exposing sensitive information through cached data.

Keylogging and the Autocorrection Database

Objective-C:

Vulnerable Code:

COPY

```
// Incorrectly attempting to disable autocorrection
[sensitiveTextField setAutocorrectionType:UITextAutocorrectionTypeNo];
```

Description: The vulnerable Objective-C code tries to disable autocorrection on a UITextField to prevent sensitive data from being stored in the autocorrection database. However, due to a bug in iOS 5.1 and later versions, the autocorrection database is still updated even when autocorrection is disabled, potentially exposing sensitive information.

Patched Code:

```
// Workaround to prevent autocorrection database update
[sensitiveTextField setSecureTextEntry:YES];
[sensitiveTextField setSecureTextEntry:NO];
```

Description: The patched Objective-C code utilizes a workaround to prevent the autocorrection database from being updated. By setting the UITextField's `secureTextEntry` property to `YES` and then back to `NO`, the autocorrection database update is effectively disabled, reducing the risk of keylogging and data leakage.

Swift:

Vulnerable Code:

```
// Attempting to disable autocorrection,
which may not work due to a bug
sensitiveTextField.autocorrectionType = .no
```

Description: The vulnerable Swift code tries to disable autocorrection on a UITextField to prevent sensitive data from being stored in the autocorrection database. However, due to a bug in iOS 5.1 and later versions, the autocorrection database is still updated even when autocorrection is disabled, potentially exposing sensitive information.

Patched Code:

```
// Workaround to prevent autocorrection database update
sensitiveTextField.isSecureTextEntry = true
sensitiveTextField.isSecureTextEntry = false
```

Description: The patched Swift code utilizes a workaround to prevent the autocorrection database from being updated. By setting the UITextField's `isSecureTextEntry` property to `true` and then back to `false`, the autocorrection database update is effectively disabled, reducing the risk of keylogging and data leakage.

Dealing with Sensitive Data in Snapshots

Objective-C:

Vulnerable Code:

```
COPY
- (void)applicationDidEnterBackground:(UIApplication *)application {
    application = [UIApplication sharedApplication];
    self.splash = [[UIImageView alloc] initWithFrame:[[UIScreen
mainScreen] bounds]];
    [self.splash setImage:[UIImage imageNamed:@"myimage.png"]];
    [self.splash setUserInteractionEnabled:NO];
    [[application keyWindow] addSubview:splash];
}
```

Description: The vulnerable Objective-C code attempts to sanitize sensitive screen content by adding a splash screen on top of all current views when the application enters the background. However, it only obscures the screen partially, leaving subviews potentially visible.

Patched Code:

```
COPY
- (void)applicationDidEnterBackground:(UIApplication *)application {
    application = [UIApplication sharedApplication];
    UIWindow *keyWindow = [[UIApplication sharedApplication]
keyWindow];
    UIView *sanitizationView = [[UIView alloc]
initWithFrame:keyWindow.bounds];
```

```
    sanitizationView.backgroundColor = [UIColor whiteColor]; // or any
other color to hide content
    [keyWindow addSubview:sanitizationView];
}
```

Description: The patched Objective-C code improves the screen sanitization process by adding a white-colored view on top of the key window, effectively obscuring all screen content. This approach ensures that all subviews are hidden, reducing the risk of sensitive data leakage.

Swift:

Vulnerable Code:

```
func applicationDidEnterBackground(_ application: UIApplication) {
    let splash = UIImageView(frame: UIScreen.main.bounds)
    splash.image = UIImage(named: "myimage.png")
    splash.isUserInteractionEnabled = false
    application.keyWindow?.addSubview(splash)
}
```

Description: The vulnerable Swift code attempts to sanitize sensitive screen content by adding a splash screen on top of all current views when the application enters the background. However, it only obscures the screen partially, leaving subviews potentially visible.

Patched Code:

```
func applicationDidEnterBackground(_ application: UIApplication) {
    guard let keyWindow = UIApplication.shared.keyWindow else { return
}
    let sanitizationView = UIView(frame: keyWindow.bounds)
    sanitizationView.backgroundColor = .white // or any other color to
hide content
```

```
    keyWindow.addSubview(sanitizationView)
}
```

Description: The patched Swift code improves the screen sanitization process by adding a white-colored view on top of the key window, effectively obscuring all screen content. This approach ensures that all subviews are hidden, reducing the risk of sensitive data leakage.

Leaks Due to State Preservation

Objective-C:

Vulnerable Code:

COPY

```
- (void)encodeRestorableStateWithCoder:(NSCoder *)coder {
    [super encodeRestorableStateWithCoder:coder];
    [coder encodeObject:_messageBox.text
forKey:@"messageBoxContents"];
}
```

Description: The vulnerable Objective-C code encodes sensitive data from a UITextView (`_messageBox`) directly using the `encodeObject` method in the `encodeRestorableStateWithCoder` method. This data may be stored insecurely on disk, leading to potential data leakage.

Patched Code:

COPY

```
- (void)encodeRestorableStateWithCoder:(NSCoder *)coder {
    [super encodeRestorableStateWithCoder:coder];
    NSString *encryptedData = [self encryptData:_messageBox.text];
    [coder encodeObject:encryptedData
forKey:@"encryptedMessageBoxContents"];
}
```

```
- (NSString *)encryptData:(NSString *)data {
    // Implement encryption logic here using a secure encryption
algorithm
    return data; // Placeholder for encryption logic
}
```

Description: The patched Objective-C code improves security by encrypting sensitive data before encoding it using the `encodeObject` method. The `encryptData` method applies encryption logic to the text data before storing it, reducing the risk of data leakage.

Swift:

Vulnerable Code:

COPY

```
override func encodeRestorableState(with coder: NSCoder) {
    super.encodeRestorableState(with: coder)
    coder.encode(messageBox.text, forKey: "messageBoxContents")
}
```

Description: The vulnerable Swift code encodes sensitive data from a UITextView (`messageBox`) directly using the `encode` method in the `encodeRestorableState` method. This data may be stored insecurely on disk, leading to potential data leakage.

Patched Code:

COPY

```
override func encodeRestorableState(with coder: NSCoder) {
    super.encodeRestorableState(with: coder)
    let encryptedData = encryptData(messageBox.text)
    coder.encode(encryptedData, forKey: "encryptedMessageBoxContents")
}

func encryptData(_ data: String) -> String {
    // Implement encryption logic here using a secure encryption
algorithm
    return data // Placeholder for encryption logic
}
```

Description: The patched Swift code improves security by encrypting sensitive data before encoding it using the `encode` method. The `encryptData` function applies encryption logic to the text data before storing it, reducing the risk of data leakage.

Format Strings

Objective-C:

Vulnerable Code:

COPY

```
NSString *userText = @"%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x";
NSLog(userText);
```

Description: The vulnerable Objective-C code directly passes user-supplied input to the `NSLog` function without proper validation or sanitization. This allows an attacker to craft a format string that exploits the `%x` specifier to leak memory contents in hexadecimal format.

Patched Code:

COPY

```
NSString *userText = @"%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x";
NSString *formattedText = [NSString stringWithFormat:@"User text: %@",
userText];
NSLog(@"%@", formattedText);
```

Description: The patched Objective-C code uses `stringWithFormat` to properly format the user-supplied input before passing it to `NSLog`. By explicitly specifying the format string and using `%@` specifier, the code prevents format string vulnerabilities and ensures that the user input is treated as a harmless string.

Swift:

Vulnerable Code:

COPY

```
let userText = "%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x"
NSLog(userText)
```

Description: The vulnerable Swift code directly passes user-supplied input to the `NSLog` function without proper validation or sanitization. This allows an attacker to craft a format string that exploits the `%x` specifier to leak memory contents in hexadecimal format.

Patched Code:

COPY

```
let userText = "%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x"
let formattedText = String(format: "User text: %@", userText)
NSLog("%@", formattedText)
```

Description: The patched Swift code uses `String(format:)` to properly format the user-supplied input before passing it to `NSLog`. By explicitly specifying the format string and using `%@` specifier, the code prevents format string vulnerabilities and ensures that the user input is treated as a harmless string.

Buffer Overflows and the Stack

Objective-C:

Vulnerable Code:

```
#include <string.h>
uid_t check_user(char *provided_uname, char *provided_pw) {
    char password[32];
    char username[32];
    strcpy(password, provided_pw);
    strcpy(username, provided_uname);
    struct *passwd pw = getpwnam(username);
    if (0 != strcmp(crypt(password), pw->pw_passwd))
        return -1;
    return pw->uid;
}
```

Description: The vulnerable Objective-C code contains a classic example of a stack-based buffer overflow vulnerability. It copies user-supplied input into fixed-size buffers (`password` and `username`) without proper validation of input size. If the input provided exceeds 32 characters, it will overwrite adjacent memory locations, potentially leading to code execution by manipulating the return address.

Patched Code:

COPY

```
#include <string.h>
uid_t check_user(char *provided_uname, char *provided_pw) {
    char password[32];
    char username[32];
    size_t pw_len = strnlen(provided_pw, 32);
    size_t uname_len = strnlen(provided_uname, 32);
    if (pw_len >= sizeof(password) || uname_len >= sizeof(username)) {
        // Handle error: input exceeds buffer size
        return -1;
    }
    strncpy(password, provided_pw, sizeof(password));
    strncpy(username, provided_uname, sizeof(username));
    struct *passwd pw = getpwnam(username);
    if (0 != strcmp(crypt(password), pw->pw_passwd))
        return -1;
    return pw->uid;
}
```

Description: The patched Objective-C code validates the size of user-supplied input before copying it into fixed-size buffers. It uses `strnlen` to calculate the length of input strings and checks if they exceed the buffer size before copying. If the input exceeds the buffer size, it handles the error accordingly, preventing buffer overflow vulnerabilities.

Integer Overflows and the Heap

Objective-C:

Vulnerable Code:

```
typedef struct Goat {
    int leg_count; // usually 4
    bool has_goatee;
    char name[32];
    struct Goat* parent1;
    struct Goat* parent2;
    size_t kid_count;
    struct Goat** kids;
} Goat;

Goat* ReadGoats(int* count, int socket) {
    *count = ReadInt(socket); // Read number of goats
    Goat* goats = malloc(*count * sizeof(Goat)); // Allocate memory
for goats
    for (int i = 0; i < *count; ++i) {
        ReadGoat(&goats[i], socket); // Read goat data from socket
    }
    return goats;
}
```

Description: The vulnerable Objective-C code contains a classic example of an integer overflow vulnerability. It reads the number of goats to process from a socket and allocates memory for goats using `malloc`. However, if the number of goats read from the socket is sufficiently large, the multiplication of `*count` and `sizeof(Goat)` may overflow, resulting in a smaller allocation than expected.

Patched Code:

COPY

```
typedef struct Goat {
    int leg_count; // usually 4
    bool has_goatee;
    char name[32];
    struct Goat* parent1;
    struct Goat* parent2;
    size_t kid_count;
    struct Goat** kids;
} Goat;

Goat* ReadGoats(int* count, int socket) {
    *count = ReadInt(socket); // Read number of goats
    if (*count <= 0 || SIZE_MAX / sizeof(Goat) < *count) {
        // Handle error: integer overflow or invalid count
        return NULL;
    }
    Goat* goats = malloc(*count * sizeof(Goat)); // Allocate memory
for goats
    if (goats == NULL) {
        // Handle error: memory allocation failed
        return NULL;
    }
    for (int i = 0; i < *count; ++i) {
        ReadGoat(&goats[i], socket); // Read goat data from socket
    }
    return goats;
}
```

Description: The patched Objective-C code checks for integer overflow before allocating memory for goats. It ensures that the multiplication of `*count` and `sizeof(Goat)` does not overflow by checking if the result exceeds `SIZE_MAX`. Additionally, it checks for a valid and non-negative count. If an overflow or invalid count is detected, it handles the error appropriately, preventing integer overflow vulnerabilities.

Client-Side Cross-Site Scripting

Objective-C:

Vulnerable Code:

COPY

```
NSString *untrustedData = [self
fetchDataFromExternalSource]; // Fetch untrusted data
[webView loadHTMLString:untrustedData baseURL:nil]; // Load untrusted
data into WebView
```

Description: The vulnerable Objective-C code fetches untrusted data from an external source and directly loads it into a WebView without proper encoding. This leaves the application vulnerable to client-side cross-site scripting (XSS) attacks, where malicious JavaScript can be injected into the content and executed within the WebView context.

Patched Code:

COPY

```
NSString *untrustedData = [self
fetchDataFromExternalSource]; // Fetch untrusted data
NSString *encodedData = [self encodeHTML:untrustedData]; // Encode
untrusted data
[webView loadHTMLString:encodedData baseURL:nil]; // Load encoded data
into WebView
```

Description: The patched Objective-C code fetches untrusted data from an external source and encodes it using a method `encodeHTML` before loading it into the WebView. The `encodeHTML` method ensures that characters are replaced with their HTML representations, preventing malicious JavaScript injection and mitigating client-side XSS attacks.

SQL Injection

Objective-C:

Vulnerable Code:

COPY

```
NSString *uid = [myHTTPConnection getUID];
NSString *statement = [NSString stringWithFormat:@"SELECT username
FROM users where uid = '%@'", uid];
const char *sql = [statement UTF8String];
```

Description: The vulnerable Objective-C code constructs a SQL statement using string concatenation, allowing for SQL injection vulnerabilities. The `uid` parameter, obtained from an external source, is directly embedded into the SQL statement without proper sanitization, making it susceptible to SQL injection attacks.

Patched Code:

COPY

```
NSString *uid = [myHTTPConnection getUID];
const char *sql = "SELECT username FROM users where uid = ?";
sqlite3_stmt *selectUid;
if (sqlite3_prepare_v2(db, sql, -1, &selectUid, NULL) == SQLITE_OK) {
    sqlite3_bind_text(selectUid, 1, [uid UTF8String], -1,
SQLITE_TRANSIENT);
    int status = sqlite3_step(selectUid);
    // Process the result
    sqlite3_finalize(selectUid);
}
```

Description: The patched Objective-C code uses parameterized statements to prevent SQL injection vulnerabilities. The SQL statement is pre-defined with a placeholder `?` for the parameterized value. The `sqlite3_bind_text` function binds the `uid` parameter to the prepared statement, ensuring that it is treated as data rather than executable SQL code, thus mitigating the risk of SQL injection attacks.

XML Injection

Objective-C:

Vulnerable Code:

COPY

```
NSURL *testURL = [NSURL URLWithString:@"http://api.nostarch.com"];
NSXMLParser *testParser = [[NSXMLParser alloc]
initWithContentsOfURL:testURL];
[testParser setShouldResolveExternalEntities:YES];
```

Description: The vulnerable Objective-C code instantiates an `NSXMLParser` instance with the option `setShouldResolveExternalEntities` set to `YES`. This configuration allows the parser to honor Document Type Definitions (DTDs) and resolve external entities, which can lead to XML injection vulnerabilities if the external entity references a malicious URL or local file.

Patched Code:

COPY

```
NSURL *testURL = [NSURL URLWithString:@"http://api.nostarch.com"];
NSXMLParser *testParser = [[NSXMLParser alloc]
initWithContentsOfURL:testURL];
```

```
[testParser setShouldResolveExternalEntities:NO]; // Ensure external
entities are not resolved
```

Description: The patched Objective-C code sets the `setShouldResolveExternalEntities` option to `NO`, ensuring that external entities are not resolved by the XML parser. This mitigates the risk of XML injection vulnerabilities by preventing the parser from fetching and including external resources.

Keychain

Objective-C:

Vulnerable Code:

```
NSMutableDictionary *dict = [NSMutableDictionary dictionary];
NSData *passwordData = [@"mypassword"
dataUsingEncoding:NSUTF8StringEncoding];
[dict setObject:(__bridge id)kSecClassGenericPassword forKey:(__bridge
id)kSecClass];
[dict setObject:@"Conglomco login" forKey:(__bridge id)kSecAttrLabel];
[dict setObject:@"This is your password for the Conglomco service."
forKey:(__bridge id)kSecAttrDescription];
[dict setObject:@"dthiel" forKey:(__bridge id)kSecAttrAccount];
[dict setObject:@"com.isecpartners.SampleKeychain" forKey:(__bridge
id)kSecAttrService];
[dict setObject:passwordData forKey:(__bridge id)kSecValueData];
[dict setObject:(__bridge id)kSecAttrAccessibleWhenUnlocked forKey:
(__bridge id)kSecAttrAccessible];
OSStatus error = SecItemAdd((__bridge CFDictionaryRef)dict, NULL);
if (error == errSecSuccess) {
    NSLog(@"Yay");
}
```

Description: The vulnerable Objective-C code adds a password item to the Keychain using `SecItemAdd`. However, it does not explicitly specify a protection attribute (`kSecAttrAccessible`). This can lead to insecure storage of sensitive data in the Keychain.

Patched Code:

```
NSMutableDictionary *dict = [NSMutableDictionary dictionary];
NSData *passwordData = [@"mypassword"
dataUsingEncoding:NSUTF8StringEncoding];
[dict setObject:(__bridge id)kSecClassGenericPassword forKey:(__bridge
id)kSecClass];
[dict setObject:@"Conglomco login" forKey:(__bridge id)kSecAttrLabel];
[dict setObject:@"This is your password for the Conglomco service."
forKey:(__bridge id)kSecAttrDescription];
[dict setObject:@"dthiel" forKey:(__bridge id)kSecAttrAccount];
[dict setObject:@"com.isecpartners.SampleKeychain" forKey:(__bridge
id)kSecAttrService];
[dict setObject:passwordData forKey:(__bridge id)kSecValueData];
[dict setObject:(__bridge id)kSecAttrAccessibleWhenUnlocked forKey:
(__bridge id)kSecAttrAccessible]; // Set a specific protection
attribute
OSStatus error = SecItemAdd((__bridge CFDictionaryRef)dict, NULL);
if (error == errSecSuccess) {
    NSLog(@"Yay");
}
```

Description: The patched Objective-C code sets the `kSecAttrAccessible` attribute to `kSecAttrAccessibleWhenUnlocked` explicitly, ensuring that the password item is accessible only when the device is unlocked. This improves the security of the stored password in the Keychain.

Encryption with CommonCrypto

Vulnerable Code (Objective-C):

```
// Using broken initialization vector
// Static IV is used, which can lead to identical blocks of ciphertext
NSData *encryptDataUsingAESWithStaticIV(NSData *data, NSData *key) {
    uint8_t iv[kCCBlockSizeAES128] = {0x00}; // Static IV
    uint8_t encryptedData[data.length + kCCBlockSizeAES128];
    size_t encryptedDataLength = 0;

    CCCryptorStatus cryptStatus = CCCrypt(kCCEncrypt,
                                          kCCAlgorithmAES,
                                          kCCOptionPKCS7Padding,
                                          key.bytes,
                                          kCCKeySizeAES256,
                                          iv, // Static IV
                                          data.bytes,
                                          data.length,
                                          encryptedData,
                                          sizeof(encryptedData),
                                          &encryptedDataLength);
    if (cryptStatus == kCCSuccess) {
        return [NSData dataWithBytes:encryptedData
length:encryptedDataLength];
    } else {
        return nil;
    }
}
```

Attack Description:

The vulnerable code snippet above demonstrates the use of a static initialization vector (IV) in AES encryption. A static IV is reused for multiple encryption operations, which can lead to identical blocks of ciphertext for different plaintexts. This makes it easier for attackers to analyze and potentially decipher encrypted data.

Patched Code (Objective-C):

```
// Using random initialization vector
// Random IV is generated for each encryption operation
NSData *encryptDataUsingAESWithRandomIV(NSData *data, NSData *key) {
    uint8_t iv[kCCBlockSizeAES128];
    int result = SecRandomCopyBytes(kSecRandomDefault, sizeof(iv),
iv);
    if (result != 0) {
        return nil; // Error generating random IV
    }

    uint8_t encryptedData[data.length + kCCBlockSizeAES128];
    size_t encryptedDataLength = 0;

    CCCryptorStatus cryptStatus = CCCrypt(kCCEncrypt,
                                          kCCAlgorithmAES,
                                          kCCOptionPKCS7Padding,
                                          key.bytes,
                                          kCCKeySizeAES256,
                                          iv,
                                          data.bytes,
                                          data.length,
                                          encryptedData,
                                          sizeof(encryptedData),
                                          &encryptedDataLength);
    if (cryptStatus == kCCSuccess) {
        return [NSData dataWithBytes:encryptedData
length:encryptedDataLength];
    } else {
        return nil;
    }
}
```

Description:

In the patched code, a random initialization vector (IV) is generated for each encryption operation using the `SecRandomCopyBytes` function provided by the `Security` framework. This ensures that each ciphertext produced is unique even when encrypting the same plaintext multiple times. By using a random IV, the security of the encryption scheme is improved, as it prevents patterns in the ciphertext that could aid attackers in cryptanalysis.

References

- iOS Application Security by David Thiel

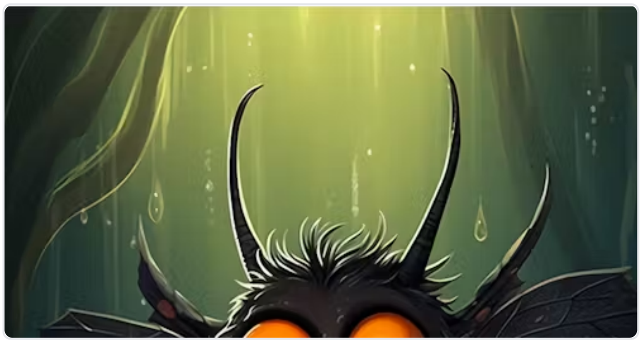
iOS Swift Objective C appsec Devops

Published on

DevSecOpsGuides Follow

MORE ARTICLES

RR **Reza Rashidi**



Defending APIs

we embark on a journey to fortify our APIs against common vulnerabilities that lurk at every stage o...

RR **Reza Rashidi**

Attacking APIs

APIs (Application Programming Interfaces) have become integral components of modern software systems...

RR **Reza Rashidi**

2FA Security Issues

What is 2FA Two-factor authentication (2FA) is a specific type of multi-factor authentication (MFA) ...