Journal of Computer Virology and Hacking Techniques (2025) 21:11
https://doi.org/10.1007/s11416-025-00556-2

RESEARCH

Defeating FIDO2/CTAP2/WebAuthn using browser in the middle and reflected cross site scripting

Christian Catalano[1] · Andrea Chezzi[2] · Vita Santa Barletta[1] · Franco Tommasi[2]

Received: 12 November 2024 / Accepted: 16 April 2025

Abstract

In our modern digital landscape, web browsers play a crucial role as gateways to large amounts of information and services. However, recent developments have demonstrated that the very features that make browsing convenient and seamless can be exploited by malicious actors through a potent threat vector known as the "Browser-in-the-Middle" (BitM) attack. Most of the Multi-Factor Authen- tication (MFA) security measures are shown to be ineffective to prevent BitM attacks. However, the FIDO2 Project that includes CTAP2 protocol that works together with the Web Authentication API (WebAuthn API) has been proven to be a virtually unattackable MFA method by current state-of-the-art BitM implementations. At least until now. This work expands the range of applica- ble scenarios where BitM attack can be used by taking its technical architecture a step further: we show how the effectiveness of BitM—used along a Reflected XSS vulnerability exploitation—can be improved resulting in the novel BitM + attack that proves to be capable of defeating any available MFA method including FIDO2/WebAuthn solutions that rely on hardware dongles and represent the only method of authentication that went undefeated by virtually any phishing attack approach to date.

Keywords Browser in the middle (BitM) · FIDO alliance · FIDO2 project · U2F · CTAP/CTAP2 · Web authentication (WebAuthn) · Multi-factor authentication (MFA) · Real-time phishing · Man-in-the-browser (MitB) · Cross-site scripting (XSS) · Malware · Malicious browser extensions · Man-in-the-middle (MitM)

1 Introduction

As highlighted by several published reports [1] on cybersecurity one of the most widely used techniques to perpetrate cyber attacks is phishing [2], and one of the most effec- tive variants of this type of attack is real-time phishing based most often on the attack pattern called Man in the Browser (MitB) [3].

Real-time phishing perpetrated through the MitB technique is a research topic of academic interest [3, 4] and it is currently analyzed by cybersecurity professionals in all of its malware variants and data breach techniques. The attacker that uses this kind of attack attempts to obtain sensitive information from a potential victim while he or she is interacting with a website or application. This technique is particularly dangerous because the attacker can manipulate web pages or applications on the go to make them more convincing and better deceive the victim.

In this kind of phishing attack, the attacker might manipulate a web page that appears to belong to a legitimate company, such as a banking institution, and then invite the victims to enter their personal or financial information by acting out many of the multi factor authentication mechanisms. Such information can then be used to steal money or perform fraudulent activities.

To protect against this type of attack, several defense mechanisms have been pro- posed over time [5] and their effectiveness and limitations have been carefully examined in literature. FIDO2 Project's CTAP2 (Client to Authenticator Protocol) along with the Web Authentication (WebAuthn) standard published by W3C stands as one of the latest

✉ Christian Catalano
christian.catalano@uniba.it

Andrea Chezzi
andrea.chezzi@unisalento.it

Vita Santa Barletta
vita.barletta@uniba.it

Franco Tommasi
francesco.tommasi@unisalento.it

[1] University of Bari Aldo Moro, Bari, Italy

[2] University of Salento, Lecce, Italy

Published online: 14 May 2025

and most commonly used strong authentication mechanisms designed to mitigate phishing in real-time on the Web.

FIDO2 [6] is designed to mitigate a range of modern cyber-attacks. In particular, it is claimed to be resistant to data breaches as servers store only public keys, and to phishing combined with Adversary-in-the-Middle because the domain is verified for each authentication. Notably, these two attacks were identified as the top cyber crimes in the Microsoft's 2022 Report [7]. FIDO2 propose itself as a paradigm shift for commonly used authentication: prior to FIDO2, passwordless authentication was primarily employed in highly secure environments, such as smart cards in government agencies and financial institutions [8]. With FIDO2, the identity community pushes towards a complete transition from the "something you know" into "something you own" (the possession factor) for a growing number of regular users worldwide.

After a close consideration of the newest web authentication technologies available today this research work aims at presenting an improved version of Browser-in-the- Middle (BitM) attack pattern which is specifically designed to invalidate the protection provided by the FIDO2 project which relies on a hardware authenticator directly connected to the user's client machine in order to confirm user's identity.

Our case study focuses on attack scenarios where it is possible to compromise web browsing security by using BitM and MitB techniques combined. MitB, in this research, is carried through by assuming that an exploitable Reflected XSS vulner- ability can be found inside any web page under a legit target service domain. So, an XSS-vulnerable website can be assumed as the target of the proposed real-time phishing attack.

To put it briefly, the core elements on which our proposed attack relies are the BitM approach combined with the MitB technique. So, we will show the joint imple- mentation of the two attacks can defeat FIDO2 and WebAuthn strong authentication protocols besides the other security methods already shown to be vulnerable in the past works [9, 10]. After a brief presentation of the related works we will proceed to describe the background that will serve to better explain our attack which is a contin- uation and improved expansion of the previous works described in [9, 10]. In addition, the present work demonstrates this technique can be applied to different vectors, since it has been shown in related works that also malicious browser extensions may be used.

2 Background

2.1 FIDO2 project and W3C WebAuthn API

To protect against real-time phishing attacks, several defense mechanisms [5] have been proposed over time. One of the most interesting proposals to mitigate this typol- ogy of attacks is definitely the adoption of FIDO2 protocols based on the passwordless concept [11]. FIDO2 stands out as a contemporary, open-source authentication proto- col that enjoys substantial backing from major technology companies (e.g., Microsoft, Google, Facebook, Apple, etc.).

Fig. 1 Some examples of commercially available hardware authenticators produced by Yubico [16] working with FIDO2 Project's protocols and WebAuthn standards

FIDO2 Project is developed by the Fast Identity Online (FIDO) Alliance [6, 12], an industry consortium composed of technology companies and other service providers. The alliance includes 42 members including Amazon, Apple, Arm, Google, Microsoft, PayPal, as well as financial companies such as American Express, MasterCard, Visa, and Wells Fargo.

The FIDO Alliance states that FIDO2 "reflects the industry's answer to the global password problem," addressing the challenges of traditional authentication in terms of security, usability, privacy and scalability. FIDO2 works alongside the WebAuthn API—designed and maintained by the World Wide Web Consortium (W3C [13, 14]).

- that are integrated into industry leader browsers, including Google Chrome and Microsoft Edge (based on Chromium project [15]), Mozilla, and WebKit. Also, FIDO2 is employed to safeguard critical targets from phishing attacks, including in Google Advanced Protection Programme. Moreover adoption of FIDO2 is rapidly expanding, propelled by the introduction of FIDO2-enabled passkeys integrated by Apple into their devices.

Consequently, it has established itself as one of the most common standards for robust, modern authentication. FIDO2 offers the versatility to function as an extra layer of security, such as alongside a password, or as a standalone authentication mech- anism. In the latter scenario, FIDO2 introduces a paradigm shift in authentication by eliminating the "something you know" element, typically represented by a password, thereby achieving the goal of passwordless authentication.

The FIDO2 protocol involves three main parties: the *FIDO server* (typically inte- grated with the *Relying Party* or simply *RP*), the *FIDO Client* (usually a web browser), and the *HW authenticator* (for instance, as shown in the Fig. 1 a USB hardware dongle). Communication among these entities relies on two standard proto- cols: WebAuthn [14], defining the interface between the client and server, and *Client to.*

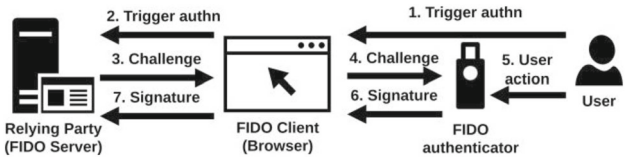

Fig. 2 Overview of FIDO2 mechanism working with WebAuthn API [11]

Authenticator Protocol (CTAP) [12] which provides FIDO2-compatible devices with an interface to external authenticators (via NFC, USB, Bluetooth, etc.).

Users can initiate two distinct FIDO2 procedures: registration and authentication. The first aims to generate cryptographic keys and associate them with the user's identity, while the latter serves to verify key possession. The protocol flow follows a straightforward challenge-response sequence, as depicted in Fig. 2. The process starts with the user's request for authentication or registration (steps 1, 2). Then, the FIDO server generates a challenge and sends it to the authenticator (steps 3, 4). To get unlocked the authenticator requests a user presence or verification check (step 5). In the case of registration, a key pair is generated, and the public key is transmitted to the server. For both procedures, a digital signature is produced and transmitted to the server (steps 6, 7). The process terminates with the server's verification of the signature. The security of FIDO2 is heavily contingent on the type (platform or roaming) and the quality of authenticators employed. A platform authenticator (PA) is an integral part of a device, like a feature of the client's operating system (e.g.: Windows, Android, Linux, macOS/iOS), while a roaming authenticator (RA) is a distinct device that can be conveniently detached, for example, a USB hardware token. Authenticator vendors are required to employ widely recognized cryptographic primitives (e.g., ECDSA); however, the security characteristics of their products may vary, including resistance to tampering. The core functions of an authenticator are to securely store private keys and to conduct cryptographic operations, such as signature generation.

Briefly, FIDO's U2F/CTAP2 standard defines cryptographic challenge and response protocols in which a dongle with a private key can prove its identity to a pre-registered website. The dongle interacts with the user's device through a physical USB port or wirelessly (NFC, Bluetooth). Such dongles are now produced by many companies, including Yubico [16] and Feitian Technologies [17].

Ultimately, FIDO2 is considered the successor to the previous authentication standards, FIDO UAF and FIDO U2F [5]. FIDO2-based solutions undergo rigorous certification to ensure that user credentials are decentralized, isolated and encrypted on users' personal devices.

As already mentioned, the FIDO2 user's private key called security key (which can also be generated from a biometric data, such as fingerprint or voice), is used to sign transactions initiated by a trusted party. Some solutions, ensure that private keys are further protected in the hardware trust zones of mobile devices, separate from the device's operating system.

2.2 Browser in the middle (BitM)

The publication of the Browser in the Middle (BitM) attack [9, 10] sparked some inter- est in cybersecurity circles [18, 19] and the general press [20, 21]. Recently (January 2023) the BitM attack technique was recognized and included in the Common Attack Pattern Enumeration and Classification catalog (list Version 3.9 CAPEC ID-701) [22] of MITRE [23]. The idea behind the attack pattern is to interpose a transparent malicious browser (the *Attacker*) between the browser of the user (the *Victim*) and the server that delivers the web service (the *Target*, e.g. a banking web application, social network, etc..) in order to intercept, record and manipulate any communication between the victim and the target of destination. Underlying the attack is a complete reversal of the classic attack paradigm. In the classic case, the attacker aims to com- promise the security of a target system. Instead, in the new paradigm, the victim is lured to visit the attacker's system and induced to use it in the belief that it is the target. It has also been shown that, when properly combined with other current tech- nologies, this technique can pose a serious threat in some of the current technological environments in which we live and operate. In this regard, a variant of the BitM attack called Persistent Mobile App-in-the-Middle (MAitM) has been described [10].

In the present research work, a new attack scenario that combines BitM with MitB properties is conceived and analyzed. To this end, a PoC will be provided demonstrat- ing how the BitM technique can be used, when combined with other vulnerabilities, to circumvent the most robust MFA authentication mechanism offered by the state of the art FIDO/CTAP2 [5, 6, 24]. The intent is to describe the attack techniques and to provide an opportunity for practitioners to better understand the risk and mitigate the impact of such an attack.

2.3 Man in the browser (MitB)

MitB is a type of cyberattack that targets web browsers and is characterized by the attacker's ability to intercept, modify, or manipulate data exchanged between a user and a web application without the user's knowledge. The "man" in this context refers to the attacker who takes control over the web browser.

Recent studies have shown that browser-based threats are evolving rapidly, exploit- ing not only user interaction but also the computational resources of end-user devices. For instance, in the MinerAlert framework proposed by Tommasi

et al. [25], the authors highlight how browser extensions and cryptojacking scripts can silently compromise a.

user's device to mine cryptocurrencies without consent—demonstrating a real-world example of stealthy, persistent threats that align with MitB tactics. Similarly, Tom- masi et al. also discuss mobile session fixation attacks in micropayment systems [26], which underline how attackers can hijack and manipulate web sessions in environ- ments with limited visibility and user control. Both examples reinforce the notion that modern MitB attacks may take various forms, from financial fraud to resource exploitation, and emphasize the urgent need for layered defenses in web environments. Man-in-the-Browser attacks are particularly stealthy and difficult to detect because they occur within the user's browser, making it challenging for traditional security measures to identify the malicious activity. The attack typically begins with the compromise of a user's device through various means, such as malware or a phishing attack. Once the attacker gains control of the browser, they can eavesdrop on the user's activities, manipulate web pages, and alter information submitted by the user. This can include stealing login credentials, credit card information, or other sensitive data. Attackers often modify web content in real-time, such as altering the details of a finan- cial transaction or changing the recipient's account number. Malicious agents can use also Man-in-the-Browser techniques to bypass two-factor authentication methods, as they have real-time access to the authentication process. These attacks can be persis- tent, meaning they can continue to intercept and manipulate data as long as the user's device remains compromised. In fact, most of the peculiarities of the MitB described.

above are intended to be leveraged to achieve the attack proposed in this paper.

Moreover defending against Man-in-the-Browser attacks often requires a combina- tion of security measures, including robust endpoint security, anti-malware software, and user education to recognize and avoid phishing attacks, however regardless of all the countermeasures that may be adopted the human factor is still the most crucial one in order to prevent security breaches by MitB.

In G. Patat et al. [3] the following statement is reported about MitB and FIDO protocols:*"[MitB] capability allows attackers to compromise the FIDO client. Thus, for instance, attackers might install malicious plugins/extensions or read browsers files, such as cookies […] Thus, by one of the MitB, the MitM and the Intrusion capabilities, attackers are able to observe the exchanged data between the authenticator and the Relying Party: users' public key, their key handle, fresh session data (e.g. challenges) and valid signatures. In this setting, the security of the U2F protocol is formally proven in A. Gonzalez* et al. [27]*and C. Jacomme* et al. [28]*even if the attackers have access to these data."*

It also turns out that not only the security of U2F protocols can be broken, but, as a consequence, it will also be shown that the newer FIDO/CTAP2 is vulnerable as well. Moreover the 2018 FIDO Security Reference document [29] admits that:*"An attacker is able to obtain malicious execution in the security context of the Relying Party client application (e.g.* via*Cross-Site Scripting (XSS)) or abuse the secure channel or session identifier after the user has successfully authenticated. This is a client side attack. Consequences: The attacker is able to control the users' session, violating Transaction Non-Repudiation."*

This means that the exploitation of XSS vulnerabilities exposed on client-side by the RP are a viable approach of attack in order to defeat FIDO/CTAP2 with.

WebAuthn API protocol. In such scenario MitB would be conveniently perpetrated and implemented by using XSS vulnerabilities exposed on client side.

In summary, a Man-in-the-Browser attack is a sophisticated threat where an attacker exploits vulnerabilities in a user's browser to gain control over online activ- ities, intercept sensitive data, and manipulate web transactions, all while remaining hidden from the user's view. In addition, as it will be demonstrated, XSS can be used as MitB in tandem with BitM attack to defeat even one of the most secure MFA methods currently available.

3 Related works

Several works already have dealt with the new paradigm for strong web authentication introduced by the FIDO2 Project. Many of these works try to point out strengths but in particular weaknesses that this new standard brings to the user's experience and his/her security. The drastic change in user's habits that FIDO2 introduces raises questions about its adaptation and usability on a large scale. For instance a study by Lyastani et al. [30], involving 94 participants revealed that while FIDO2 passwordless authentication was well received, concerns about usability, such as recovery from a lost authenticator, persisted. Similarly, Owens et al. [31] examined user perceptions of passwordless authentication using RAs (Roaming Authenticators) and reported user concerns related to availability, account recovery/backup, and setup difficulties.

The research paper detailed in E. Ulqinaku et al. [4]. analyzes the effectiveness of FIDO2 protocols in preventing real-time phishing attacks. In particular, it focuses on possible downgrade attacks against FIDO protocols, which could circumvent their protections pushing the users by any kind of social engineering trick to use a fall- back authentication option in order to get authenticated and gain access to restricted data and resources. All in all, in-depth analysis of vulnerabilities in FIDO proto- cols revealed that these

standards may not be able to completely eliminate real-time phishing attacks. In particular, attackers could use social engineering techniques to manipulate authentication messages sent between the user and the server, forcing the user to provide their login credentials without being protected by FIDO2.

A few researchers conducted several experiments to demonstrate the feasibility of these attacks [27, 28]. For example, they showed that it is possible to trick users into providing their credentials through a counterfeit website using a combination of phishing techniques and manipulation of authentication messages. In addition, the researchers examined several countermeasures to mitigate these attacks, including the use of extended Secure Sockets Layer (SSL) certificates and the adoption of multi- factor authentication protocols. However, they concluded that these solutions may not be sufficient to completely prevent social engineering downgrade attacks against FIDO protocols.

In [28] a valid formal analysis of multi-factor authentication protocols is conducted. It shows, by using formal and consistent definitions and notations what kind of attacks may defeat different MFA protocols including FIDO/CTAP2. It also claims that it is possible to bypass FIDO/CTAP2 protection in attack scenarios under certain condi- tions that involve malicious code to be run on the client machine. Instead, in [27] the.

authors analyze of the YubiKey dongle (produced by Yubico) and a specific version of the YubiHSM protocol. However such analysis was conducted by focusing on the inner cryptographic specifications of the USB dongle that implements the FIDO2 Project authenticator hardware and not considering the possible security issues that may arise with attacks aiming at other parts of the FIDO protocol such as the WebAuthn API, the browser involved in the authentication and the communication channels between server and clients ends crucial to get the authentication process working.

Papers from C. Catalano et al. [9, 10] showcase the innovative Browser-in-the- Middle (BitM) and Persistent MobileApp-in-the-Middle (MAitM) attacks that can defeat the majority of the MFA mechanisms except for the FIDO/CTAP2 one. How- ever this work stands as prosecution of the research conducted by [9, 10] in order to overcome their limitations and be able to defeat FIDO/CTAP2 protection too.

In works from J. Reynolds et al. [32]. and M. Kepkowski et al. [11]. Usability issues of this FIDO2 solution are pointed out where practicality of the usage of hardware dongle to guarantee security is questioned. In fact, it is discussed how the introduction of such hardware dongles poses new problems to the security of the users such as the possibility of the dongle theft and to the service scalability for the companies that would need to guarantee as certain degree of recovery capability in case the authenticator dongle is used as the only method of access and it gets whatever lost or destroyed.

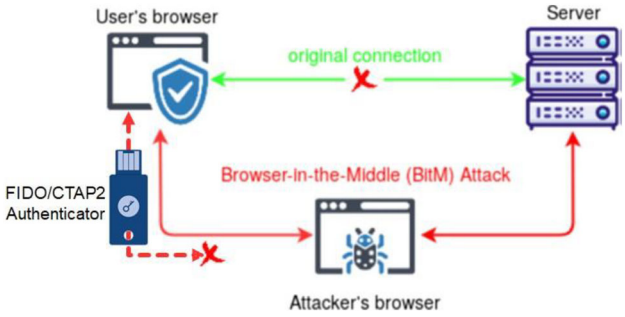

Fig. 3 Regular BitM design fails to phish FC2W authentication due to absence of a dedicated communication channel that supports WebAuthn API malicious data exchange between the user's browser and the BitM. In fact the connection between user's browser and BitM provides to the victim no more than the real-time video feed of the target webpage and standard mouse/keyboard functionality for the I/O. Hence, FIDO authenticator functionality, in this case, is out of reach of the BitM. [9]

Chezzi et al. [33] present a variant of BitM + attack, using an approach based on malicious browser extensions instead of XSS vulnerability exploitation.

In summary, the research so far suggests that although FIDO protocols are useful in preventing phishing attacks, they may not be sufficient to completely eliminate real-time phishing attacks. The researchers suggest further technological and security developments to address this growing threat. On the other hand, the new attack proposed in this paper represents a new example of what can be done in order to defeat such strong authentication mechanisms as FIDO2 Project.

4 The idea behind the proof of concept

For simplicity sake, from now on, the authentication mechanism known as "FIDO/C- TAP2 with WebAuthn API" might be called just "FC2W".

4.1 Why regular BitM attack does not work with FC2W

Before presenting the new attack, we start by acknowledging why a regular BitM attack (as shown in the Fig. 3) is ineffective to defeat the strong authentication mechanism implemented with FC2W. To this purpose, we refer to Sect. 7.1 in F. Tommasi et al. [10] entitled "MAitM effectiveness assessment" where the set of conditions required by such kind of attacks to succeed (MAitM or equivalently BitM) is discussed. Long story short, we can state that a BitM attack that targets an authentication method that requires the physical attachment—on the client machine that is used by

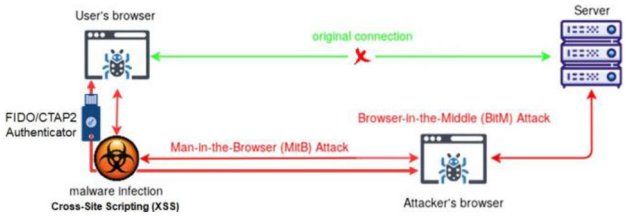

Fig. 4 Idea behind our proposed attack based on BitM and MitB. In contrast to the regular BitM approach in [9, 10], and illustrated in Fig. 3, this new attack design implements also an additional data connection via HTTP between the FIDO hardware authenticator attached to the user's machine and the attacker's BitM machine

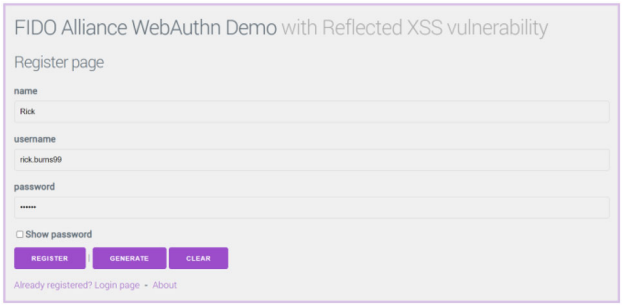

Fig. 5 Registration page of the target RP

the user to perform authentication—of anything more than mouse and keyboard devices, is doomed to failure.

This means that to overcome this limitation in the BitM attack we need to establish between the user's client and the attacker's BitM not only a connection that provides.

I/O such as the full browser web view, mouse and keyboard—as it happens so far—but also an additional communication channel that makes possible the exchange in both directions of the WebAuthn API data between the Relying Party server (and in this case the BitM itself) and the client computer—to which the authenticator is attached. In fact, without the payload (the challenge) generated from the Relaying Party and received by the BitM (which sits *in the middle* between the Relaying Party and the authenticator), it is impossible for the authenticator to generate and sign the assertion needed to complete service authentication. At the same time, it is impossible for the BitM with just the possession of the challenge—received from the Relying Party—to generate and to sign the assertion needed to finalize service authentication if there is no viable way to reach the hardware authenticator (only reachable by the victim's computer).

This is the reason why the key point of our work is the design and the working implementation of an additional malicious communication channel which will enable the transparent WebAuthn API data traffic between the client computer (that it is required to be compromised with a MitB attack) and the BitM. In Fig. 4 a synthetic scheme of our attack idea is illustrated.

4.2 Targeting the W3C WebAuthn API

The proposed attack does not target the cryptography technology that ensures the FIDO hardware authenticators security itself (e.g.: USB dongles) but operates on the communication between the Relying Party (the server which provides a web service account access e.g. Google Gmail, Facebook account etc.) and the client's computer. Such communication relies on WebAuthn APIs, which have been standardized and implemented in virtually all modern browsers in order to enable strong authentica- tion. Specifically, WebAuthn implements an extension of the W3C's more general Credential Management API, which aims at formalizing the interaction between web- sites and web browsers when exchanging user credentials. The Web Authentication API [34] extends the Credential Management navigator.credentials.create() and navigator.credentials.get() JavaScript methods so they accept a publicKey parameter. The create() method is used for registering public key authenticators as part of associating them with user accounts (possibly at initial account creation time but more likely when adding a new security device to an existing account) while the get() method is used for authenticating (e.g. when logging in) [34, 35]. However, for the purposes of our attack, named *BitM* + from now on, we will focus on the get() method in particular.

4.3 The WebAuthn API authentication operation

WebAuthn API implements two use cases: the registration with create() and the authentication with get(). Just once, at first, user registers and associates his/her hardware authenticator with an account that implements FC2W strong authentica- tion. In the following, each time the need to authenticate into the account arises, all the user has to do is nothing more than to attach his/her authenticator to the client computer and to use a gesture (e.g. a key press) to complete the login.

4.4 Targeting a reflected XSS vulnerable RP's website

In order to carry on the demonstration of our attack we need a target website (as shown in Figs. 5, 6 and 7) that implements WebAuthn API as MFA method and at the same time also exhibits a Reflected XSS vulnerability. Such scenario is obtainable if we can get done at least one of these two things:

To find in the wild an RP website that implements FC2W protection and is vulnerable to Reflected XSS attacks, enabling JS injection by the attacker.

Fig. 6 Login page of the target RP

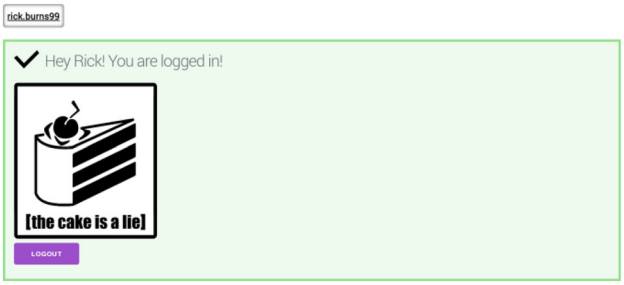

Fig. 7 Private area once the user has logged in

To build a demo RP website with the requested features i.e.: having FC2W imple- mentation and a Reflected XSS vulnerability deliberately put in place for the attack demonstration purposes.

In this work, we are dealing with the second possibility, which proves to be more practical. So we introduce a simple demo web application with a web server—imple- menting both regular password and WebAuthn API as authentication factors—that exposes to the client a registration page, a login page and a reserved user's profile page once the client has successfully logged in. Such web application not only implements password and WebAuthn API as authentication methods but also has been deliber- ately provided with a Reflected XSS vulnerability, exploitable by the attacker using an URL parameter as an entry point to inject malicious JS code.

The code base implementing both front-end and back-end of the demo web appli- cation—representing the target RP in our work—has been borrowed by the original.

GitHub repository "webauth-demo" published by Fido Alliance and other contribu- tors [36–38]. However the original repository's code has been enhanced and modified by us in order to render it more suitable for our demonstration purposes.

In our attack scenarios the RP website will be considered reachable at the URL "https://fido.site.demo/".

4.5 Principle of the attack

As detailed in [9, 10], a normal BitM attack is able to defeat normal one factor (based on "what I know", i.e. passwords, PINs, etc.) and two factors authentication methods (namely those based on "who I am", i.e. external messages, fingerprints, face detection etc. or based on "what I own", i.e. OTPs sent via SMS or emails etc.). At this regard, Table 3 at Sect. 5.2 in [18] shows state of the art of BitM attacks' effectiveness against most of authentication methods available today, including FC2W.

What can be done when a device is physically connected to the client computer making, by FC2W, BitM insufficient to overcome the security barrier? To succeed with an attack, some sort of MitB must be added to BitM. In our case MitB will be implemented by the exploitation of an XSS vulnerability inside Relying Party website. To better explain our proposed attack and highlight its underlying rationale, we refer to three user authentication scenarios with FIDO/CTAP2 with WebAuthn API.

(FC2W) in action:

1. *Scenario 1*: Legit user authentication using FC2W without any attack in progress.
2. *Scenario 2*: User authentication using FC2W attacked by regular BitM technique that proves to be ineffective in compromising user's security.
3. *Scenario 3*: User authentication using FC2W attacked by the proposed BitM + that proves to be successful in compromising the user's security.

4.5.1 Scenario 1

The first and simplest scenario relates to how FC2W authentication is supposed to work under normal circumstances. Figures 8 and 9 show its operations sequence dia- gram and block diagram. In particular, the steps shown in Fig. 9 can be described as follows:

The legit user wants to authenticate to a Relying Party's web service, so he/she inserts username and password credentials that are sent by the client's browser to the server (step 1 in Fig. 9 / Seq.1 in Fig. 8).
The Relying Party's server acknowledges the received credentials. (Seq.2 in Fig. 8)
If the credentials are correct, Relying Party generates a new challenge. (Seq.3 in Fig. 8)
The challenge (which is basically a token string generated with specific crypto- graphic functions) is sent to the client browser (Seq.4 in Fig. 8) together with other data specified in the WebAuthn API, including the mandatory *rpId* (Relying Party ID). The rpId, for instance, is a string-type JSON

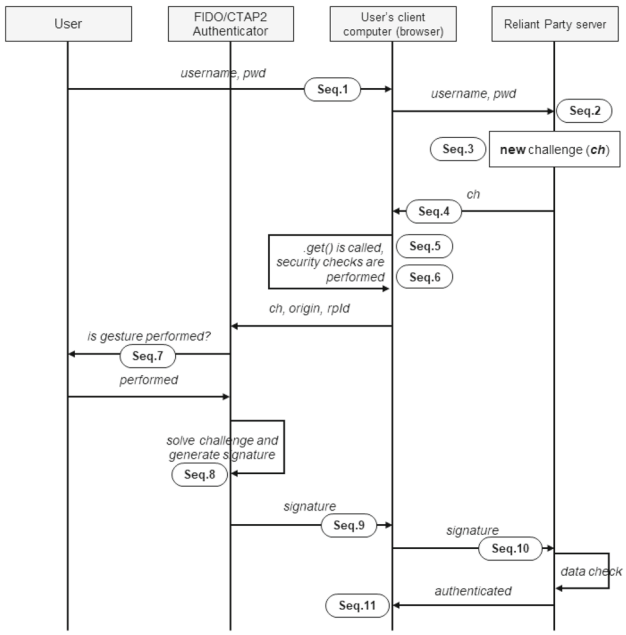

Fig. 8 Sequence diagram of regular user authentication with FC2W and no attack in progress

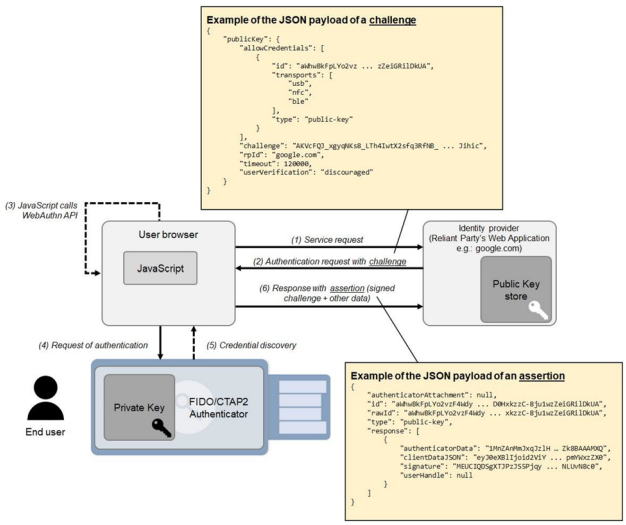

Fig. 9 Block diagram of regular user authentication with FC2W and no attack in progress. Authen- tication steps are numbered. Examples of challenge and assertion JSON data objects are shown

field that must contain the domain name of the Relying Party (e.g.: "google.com" for Google Services).

Once the payload that includes the challenge and rpId data is received by the client (step 2 in Fig. 9), the method navigator.credentials.get() of WebAu- thn API is called by the login web page (step 3 in Fig. 9 / (Seq.5 in Fig. 8). This method takes as argument the payload containing the challenge token, the rpId and other optional data.

At this point two crucial security checks take place (Seq.6 in Fig. 8), we will call the SC1 and SC2. The first one (SC1) is performed by the browser itself, that checks if the rpId matches the origin (source URL string) of the page and meets all the requirements as prescribed in the documented specifications of W3C WebAuthn API [13].

If a mismatch occurs then the authentication process terminates with a failure.

If SC1 is passed, the web browser goes to SC2 and calls the WebAuthn API method implemented in the host OS (e.g. Windows, Linux, Android, macOS, iOS) which is committed to trigger the system-wide administration-level authentication window that asks the user to attach the hardware authenticator to the computer and to perform the confirmation gesture (step 4 in Fig. 9).

After the user attaches the hardware dongle the WebAuthn API of the OS checks whether the hardware authenticator is the legit one and whether it has been actually registered to the specified Relying Party (this is SC2 and it is done by checking once again the rpId field).

If the hardware dongle is found to be correct, then SC2 is also passed and the OS will ask the user to perform the confirmation gesture (Seq.7 in Fig. 8). At this point, the authenticator proceeds to get the challenge from the browser through OS WebAuthn API in order to sign it and generate the payload with the assertion required for authentication.

The assertion (that is nothing but cryptographic string data)—that can be cor- rectly generated and signed only by the right authenticator (Seq.8 in Fig. 8) with the right challenge—is gathered by the OS and passed to the web browser (step 5 in Fig. 9 / Seq.9 in Fig. 8).

Now the navigator.credentials.get() method called in the browser, gets the response of the user's authenticator it was waiting for and it is finally able to return as result the JSON data object containing the assertion and the signed challenge just obtained from the authenticator.

Now, the JSON payload can be transmitted back from the client's browser to the Relying Party's server that can check its validity (step 6 in Fig. 9 / Seq.10 in Fig. 8).

If the assertion is evaluated as valid, then the authentication phase is completed and the user is granted an authenticated account session (Seq.11 in Fig. 8).

4.5.2 Scenario 2

The second scenario describes a regular BitM attack against FC2W and the consequent failure of the attack. Such scenario is taken into account in order to explain why a regular BitM attack is insufficient to break the defence built by FC2W authentication and why a more sophisticated approach has to be conceived.

Let there be three main actors: V (the victim's browser), B (the attacker's machine implementing regular BitM) and RP (the Relying Party server which provides a web service e.g.: Google GMail).

As preliminary conditions, let's start by assuming that the victim has just run into a malicious webpage that exposes the BitM attack. This means that the victim is tricked into believing that he/she is directly connected to RP's authentication webpage while instead he/she is just remotely interacting with the webpage instance running on the BitM attacker's machine.

At this point the following events occur:

Once V is on the BitM malicious page he/she inserts via keyboard the users credentials that are hence provided to B.
Now B, in possession of the credentials, asks RP (Relying Party) for authentica- tion.
Since the victim's account is protected by FC2W the RP generates a new challenge.
The RP sends the challenge and the rpId not directly to V but to B which is the BitM that is—in fact—in the middle.
Now that B has the new challenge and rpId the RP's front-end webpage calls on B the navigator.credentials.get() method with the challenge as argument.
This implies that the browser in B (that is BitM's browser) will perform the secu- rity check between rpId© and webpage origin (SC1). This check is passed because the BitM in B is actually navigating the legit RP's webpage which exhibits an origin that matches the one that is indicated in the rpId provided by RP.
Now the browser in B will pass the challenge and rpId to the host OS on the same machine in order to get in touch with the hardware authenticator through WebAuthn API implemented on the OS.
At this point host OS on B will prompt system level authentication window that asks the user to attach the hardware dongle in order to perform SC2 check on the rpId and then generate assertion and signature based on challenge.
But this is not possible: In fact the hardware dongle (authenticator) is physically owned by V and the host OS asking for HW authenticator attachment is actually B's, potentially residing in another continent with no way to reach required V's authenticator. In such circumstance, it means that SC2 cannot even be performed in the first place, since V's authenticator is missing and out of reach from the B's perspective. So SC2 fails (and with it the entire authentication procedure) before it can even start in the first place.
But, once again, let's concede that the attacker can actually craft a counterfeit HW dongle that, once hacked by the attacker, contains a fake registration to the target rpId. Under these forced circumstances, SC2 could be passed.
Now that also SC2 check is behind us, ultimately it still remains the fact that the counterfeit HW dongle in attacker's possession could never replicate the cor- rect private key contained exclusively in V's dongle. Only the private key in V's dongle can actually sign and generate the assertion required by RP's server for a successful authentication. This means that, excluding other attack approaches based on the discovery and exploitation of potential hidden weaknesses inside the FC2W dongles' cryptographic security algorithm (which is based on asymmetric keys), it is not possible—under regular BitM attack conditions—to overcome that obstacle.

Hence, as it has been described, there are multiple reasons why a regular BitM attack would never work against FC2W authentication mechanism and those include:

1. SC1 fails (cross-domain error, solvable with some minor modifications to the regular BitM approach).
2. SC2 fails (could by bypassed and eluded with hardware or software dongle emulation and still using regular BitM approach).
3. RP-related private key on hardware authenticator is unreachable (unsolvable under regular BitM attack architecture conditions).

4.5.3 Scenario 3

Now, after we discussed in Scenario 1 how FC2W normally works and in Scenario 2 why regular BitM attack alone (as detailed in [9, 10]) is not enough to defeat FC2W strong authentication, we are in a better position to figure out what it takes to design a successful attack against FC2W. Scenario 3 basically describes the idea and the principles behind the new "BitM + " attack here proposed, which makes use of a BitM component that works together with a MitB one.

Scenario 3 almost entirely focuses on the same three actors and preliminary conditions already described in Scenario 2.

However a few substantial differences are here introduced.

Let there be three main actors: V (the victim's browser compromised with MitB by leveraging XSS), B (attacker's machine implementing a new BitM, enhanced in order to handle network connection with MitB in V) and RP (the XSS-vulnerable Relying Party server that we previously introduced).

Among the preliminary conditions, it will be assumed that the victim has just run into a malicious webpage that exposes the BitM attack just as it happens in Scenario 2. However a key difference in the present scenario is in that—thanks to a malicious XSS payload enabling a MitB attack in V—it is now possible to use a legit URL (exhibiting same origin domain) of the RP login webpage as a vector of the improved BitM/MitB attack ("BitM + "). The key point is in that, when a same ori- gin URL (belonging to RP web service, e.g.: "https://fido.site.demo/?xssParam =

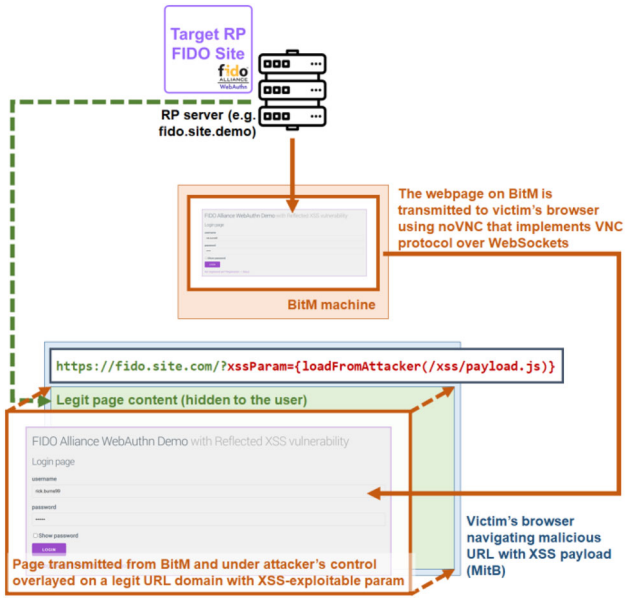

Fig. 10 When the victim navigates *xURL* the malicious JS code payload is injected inside the RP client webpage by exploting XSS vulnerability. Such code injection is responsible of overlaying the page view transmitted from BitM and providing control of client WebAuthn API functionalities to attacker. Note that a similar result can be achieved by infecting the victim client with a malicious browser extension, having a yet more powerful vector of attack in terms of flexibility and persistence

*{*loadFromAttacker(/xss/payload.js)*}*") with a XSS-exploitable parameter attached to it, is navigated by the victim, the malicious JS payload, inside the xssParam content, proceeds to hide the original content of the actual login webpage and overlays it with the webview transmission of the login webpage controlled by B's BitM. We refer to such malicious Reflected XSS payload as "*xssPayload*". The mali- cious parameter is used as the XSS exploit entry point which enables the attacker to inject arbitrary JS code inside the RP domain and execute it. We will call *xURL* such kind of malicious URL. A real xURL looks like the following:

https://fido.example.target.site: PORT/
?s = < img hidden src = x onerror = ".
{let svr = 'https://attacker.site:PORT '; localStorage.setItem('svr',svr); let j = document.createElement('script'); j.src = svr%2B'/xss/payload.js'; document.body.appendChild(j)}" >

But for practical reasons we will just omit its integral form from now on. Figure 10 illustrates the concept just introduced.

This not only means that the victim is remotely interacting with the webpage actually served by the BitM attacker's machine (B) but, by the xssPayload, the victim is also doing so with a legit URL from RP displayed in the V browser's address bar. We'll prove below how the method of implementing a MitB by xssPayload makes it possible for the attacker to overcome the very obstacles met in Scenario 2, which were at the root of the failure of the regular BitM approach. Note that the stage thus reached is akin to the consequences of a Reflected or Persistent XSS vulnerability exploit where an arbitrary payload from the attacker can produce a malicious JavaScript code injection into a page coming from a trusted source.

However, it is safe to say that a similar MitB attack can be also perpetrated with malicious extensions installed in victim's browser that enable cross-site scripting inside the RP's target authentication webpage, thus providing the attacker with the chance to remotely execute arbitrary JavaScript code in any page of V's browser. Chezzi et al. [33] presented a work on this alternative approach.

It is also worth pointing out that the attack approach presented in Scenario 3 demonstrates a method of leveraging the hardware (HW) authenticator against the unsuspecting victim. In fact, the additional communication channel introduced by BitM + is essential for establishing a malicious connection between the victim's physically-owned HW authenticator and the attacker-controlled BitM machine. While the HW authenticator remains part of the authentication process, its role is effec- tively undermined from a security standpoint. This mechanism bears resemblance to Microsoft's RDP authentication features, as described in the first paragraph of Sect. 4.6.

Figures 11 and 12 show the sequence diagram and block diagram of Scenario 3. In particular, the steps shown in Fig. 12 are included and referenced in the following description of the present scenario:

The victim using browser V navigates the attack vector URL, that we call *xURL* (e.g.: https://fido.site.demo/?xssParam = loadFromAttacker(/xss/payload.js)). When such xURL is navigated, the JS script (*/xss/payload.js*) served by the attacker machine B is injected and exe- cuted inside RP page on V's browser. Such script is committed to transmit and overlay to the victim's navigated page the webview of the RP's authentication page actually residing in B's BitM (as showed in Fig. 10). Hence the victim is actually interacting with the RP's authentication page forwarded and controlled by BitM, and when he/she inserts the credentials via keyboard, they are actually collected by the BitM webpage under B's control.

At this point B, through the gained credentials, asks RP for authentication (step 1 in Fig. 12).

Since the victim's account is protected by FC2W, the RP generates a new challenge.

The RP sends the challenge and the rpId not directly to V but to B on which is BitM running and which is—in fact—in the middle between V and RP (step 2 in Fig. 12).

Now B has the new challenge and the rpId which are packed in a JSON object that we call chObj (chObj can also contain

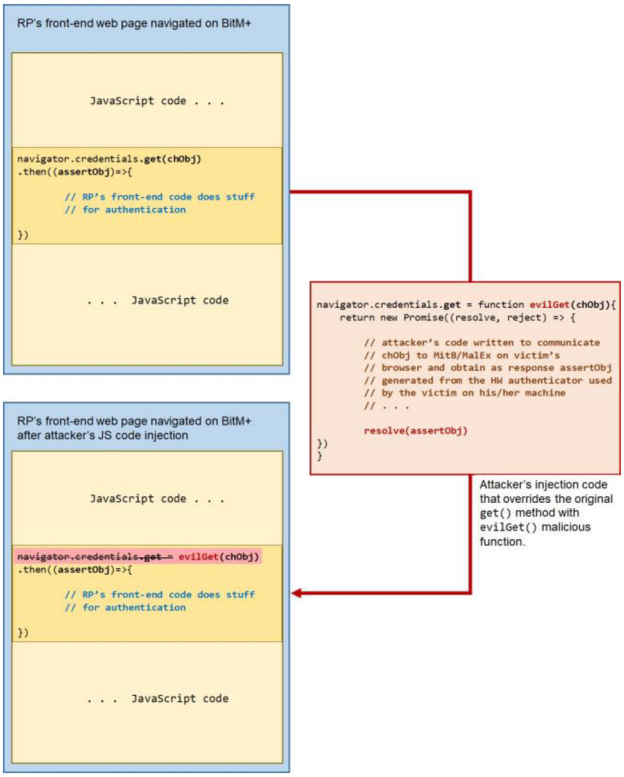

Fig. 11 JS code injection into RP's front-end authentication page opened in BitM + in attacker's B machine: the attacker injects the code that overrides the original get() method with evilGet() malicious function. evilGet() still takes in input chObj from the RP and gives in output an assertObj. The difference is in the fact that the assertObj, in the case of original get() method, is collected from HW authenticator attached to the user's local machine, in the case of the evilGet() function the assertObj is obtained from the HW authenticator attached to the victim's remote machine infected by MitB/xssPayload. This figure shows how steps 3 to 7 in Fig. 12 are implemented

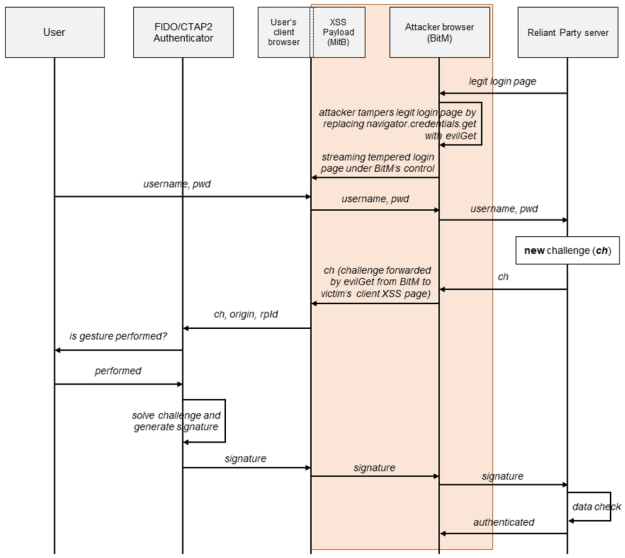

Fig. 12 Sequence diagram of a successful attack that defeats FC2W using the proposed approach. The light orange area highlights where the BitM + attack components (the MitB malicious xssPayload injected in RP page on user's browser and the BitM in attacker's machine) place themselves in order to compromise the FC2W user authentication security

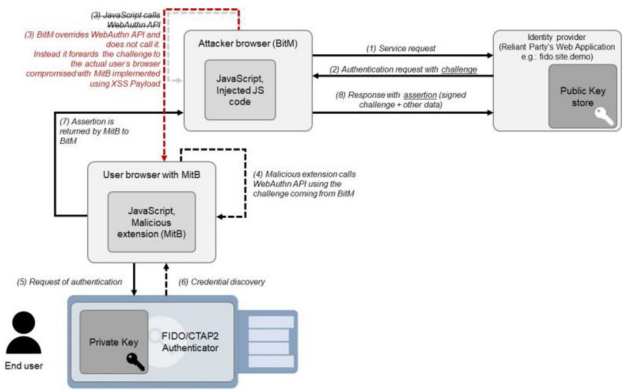

Fig. 13 Block diagram of a successful attack that defeats FC2W using the proposed approach. Authentication steps are numbered and briefly commented

other optional data as specified in WebAuthn standard). Then, the RP's authentication webpage on B's BitM *does not* call, as it should normally do, the navigator.credentials.get() method (crossed out step 3 in Fig. 12), but the BitM is specifically modified in order to override such get() method present on the original RP's authentication webpage (Fig. 13) and replace it with a custom JavaScript function—that we will refer to as evilGet(). So evilGet() in place of get() is called. The function evilGet(), once called, stores in the BitM the JSON object containing challenge and rpId data (chObj). Such chObj is made available on the Internet via a simple web server (that exposes HTTP REST APIs coded by the attacker) running on B. It is important to note that the replacement of get() method implementation does not change the purpose of the method itself, but changes *how* that purpose is achieved. When the original get() method is called, it directly provides the HW authenticator with the chObj content and waits for assertion and signature to be generated and returned: in other terms the use of such method is nothing more than to obtain the data required for authentication from the HW dongle. When no attacker interferes in the middle, everything happens in the legit user's machine (V). On the other hand, the method evilGet()—which overrides original get()—is a function that collects chObj, makes it available on the Internet and waits for a response from an external network actor in order to get the expected authentication data. In some way we can say it "publishes" chObj online. This principle is illustrated in Fig. 13.

So the key thing here is that the original (and legitimate) method (execut- ing in V) expects response (i.e. the

signed ChObj) directly from local trusted HW authenticator, meanwhile the modified—and malicious—method (actually executing in B) awaits the same from *someone else* on the Internet (spoiler: the "someone else" will obviously be V's browser where xURL is opened and xssPayload is running).

This aspect is illustrated in Fig. 12. In particular, in such figure, it is shown a crossed-out Step 3 (representing original.get) and a red Step 3 (representing malicious.evilGet that replaces.get functionality).

Keep in mind that since everything in B—especially the BitM—is under the attacker's full control, it is trivial for the attacker to craft such modifications to the BitM WebAuthn APIs using JavaScript code injection.

We'll see below how this will be accomplished.

Once chObj containing challenge and rpId is exposed online by B, then the V actor comes back into play. Let's recall that V—in the present Scenario 3—is represented by the victim's client browser that has been infected with MitB/xssPayload. MitB/xssPayload is designed to operate so that, while the xURL is navigated to and opened in V, it also polls in the background the HTTP server set up by B, to check whether a new chObj is available there.

Such a chObj will be eventually found at the HTTP server as the event chain unfolds (B will have put it there as it received chObj from RP).

When xssPayload, unwittingly executed by V gets chObj from the web server, it can make a successful call to the original local WebAuthn API navigator.credentials.get() method and give chObj as argument to it (step 4 in Fig. 12).

Now, the first security check (SC1) takes place and it is passed, since the rpId in chObj matches the domain of the webpage thanks to the fact that—as above clarified—all the arbitrary JavaScript code involved in V (including the get() call) can be run by xssPayload directly in the xURL's webpage context which is considered directly originated by the trusted RP (hence no domain mismatch or cross-origin error is detected).

Once SC1 is passed, the infected browser in V calls WebAuthn API method implemented in the V's local host OS which will trigger the system-wide administration-level authentication window (Fig. 14 shows an example of it) that asks the victim to attach the hardware authenticator to the computer and perform the confirmation gesture. Note that the elevated dialog window (i.e. a dialog executed with administrative privileges) is spawned directly by the OS on the victim's local machine V and it is not part of the BitM webview transmission (step 5 in Fig. 12). Furthermore the window tells the victim that the origin of the authentication request is the V's trusted local browser and not the remote BitM, making the process completely transparent and seamless to the victim and causing Scenario 3 to appear indistinguishable from Scenario 1 in terms of user experience.

After that, the unaware victim attaches the HW authenticator, the WebAuthn API of the OS checks whether the hardware authenticator is the legit one and whether it has been actually registered to the specified Relying Party (this is the security check SC2 and it is performed by examining the rpId field in the chObj).

If the hardware dongle reveals to be correct, also SC2 is passed and the OS will ask the user to perform the confirmation gesture (e.g.: a finger tap or button press). After the victim's gesture, the HW authenticator proceeds to get the challenge from the browser through OS WebAuthn API in order to sign it and generate the payload with the assertion data required for authentication.

The assertion is gathered by V's OS which in turn passes it to the infected web browser (step 6 in Fig. 12).

At this point the original navigator.credentials.get() method that was called in the browser by the xssPayload, awaiting for the response of the victim's authen- ticator, is finally able to return as result the JSON data object—that we will call "assertObj"—containing the assertion and the signed challenge obtained shortly before from the HW authenticator.

Now xssPayload in V has assertObj and can send it via REST API to the B's attacker web server.

The B's web server receives assertObj (step 7 in Fig. 12) and the method evilGet(), that was waiting for response from V, can finally return the call with the assertObj data.

Now that the JSON payload of assertObj—containing all the information needed for RP to grant authentication—has arrived to B, it can be sent as response from B to the Relying Party's server that can check its validity (step 8 in Fig. 12).

The assertion is evaluated as legit and then the authentication is finally granted. Hence, BitM in B is now successfully provided with an authenticated account session that was protected with FC2W. The victim will remotely and unknowingly interact with his/her account session only through the BitM in B under attacker's control. In other words the user (victim) session is persistently opened in B's BitM not in V's web browser, and the victim is just navigating—via BitM + attack—in his/her session available in B attacker's web browser.

4.6 Observations

4.6.1 Similarities and differences with Microsoft's Remote Desktop Protocol (RDP) implementation

It can be noted that the entire BitM + attack presented in Scenario 3 is function- ally similar to something that already exists today and it is actually in operation in some cases, even if for legitimate purposes. In fact, for instance, the native appli- cation "Remote Desktop Connection" for Microsoft Windows (also known with its former name "MS

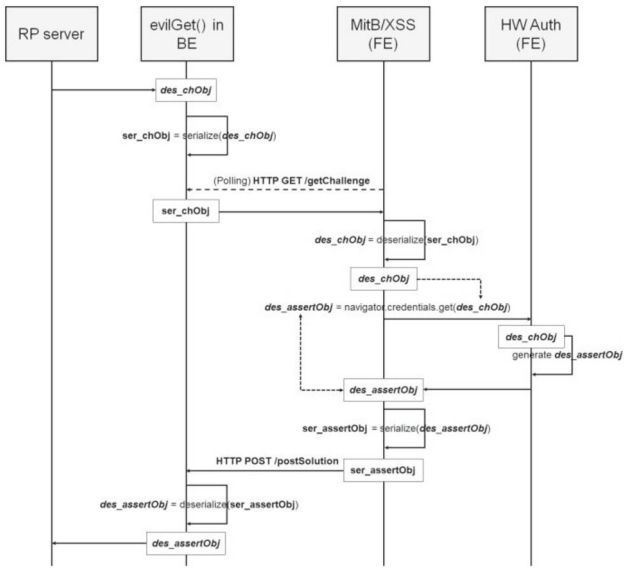

Fig. 14 Sequence diagram of chObj and assertObj undergoing serialization/deserialization opera- tions during the BitM + attack (Scenario 3)

Terminal Services Client") has recently (2022) implemented and documented [39]*"the Remote Desktop Protocol (RDP): WebAuthn Virtual Channel Protocol (MS-RDPEWA) which provides a way for a user to do WebAuthn operations over the RDP protocol. It enables a server to send WebAuthn request to a client, the client can then use this request to talk to authenticators (platform as well as cross- platform) and reply with the response"*. That means that this commercially available.

protocol is normally adopted to enable users who work with remote desktop sessions to get authenticated on their remote machines to a service protected with FC2W strong authentication, by still using their FIDO hardware dongle that keeps working locally attached to the machine next to them. In other words, MS-RDPEWA protocol implements a bridge connection (in fact described as a "Virtual Channel") for FC2W between the local machine that can physically access the hardware dongle and the remote machine that needs the dongle in order to be able to authenticate.

Our proposed BitM + attack does functionally the same thing, but focuses on its potentialities in phishing attacks scenarios.

The crucial technical difference between MS-RDPEWA and our BitM + attack design is in that our solution—though it provides the same functionality as the MS- RDPEWA protocol—it is not relying on any native application intentionally installed or directly built-in in the OS. Still instead, it merely relies on a webpage accessible by practically any web browser and on a XSS vulnerability or on a (malicious) extension installed into it.

Note that an RP (Relying Party) service provider could, if it only wanted to, provide by itself a remote FC2W authentication implementation entirely reliant on a webpage in the browser and nothing else. That means that the use of an extension or JS injection via XSS would become unnecessary for those non-malicious FC2W's use cases that are specifically laid down by the RP.

In any case, it can be said that such procedures—if implemented—have phishing attacks as their most obvious practical application, and that is, in fact, the main focus of the present work.

4.6.2 Cross-platform-ness

Another implication of our attack design is that it is cross-platform from an OS stand- point: on the client/victim side it relies exclusively on web browser technologies and HTML/JavaScript code that follow the same standards regardless of the OSes on which they sit. We can say that web browser in this case acts like a virtual machine on which the same malicious code can be run regardless of the OS platform. How- ever, it could be noted that over time this is proving to be increasingly true for more and more legit applications too where the "hybrid app" approach caught on in app development and software engineering fields in general. We can define hybrid app a type of mobile application that combines elements of both native and web applications. It is developed using web technologies such as HTML, CSS, and JavaScript, and then wrapped in a native container (a browser) that allows it to be installed and run on mobile devices like a native app. Basically, hybrid apps leverage web technologies for cross-platform compatibility, enabling developers to write code once and deploy it on multiple platforms (e.g., iOS, Android). After such brief description, similarities emerge between the hybrid app design approach and the BitM + attack, in particular, with respect to the part of the attack represented by the xssPayload executed on the victim's browser and potentially running on any OS platform.

4.6.3 Different ways to implement MitB

It is also worth noting that the effects of a MitB attack, achieved with the exploita- tion of a Reflected XSS vulnerability within the RP's (Relying Party) authentication webpage, can also be replicated by installing a malicious extension in the browser.

However, in both cases some requirements must be met: if a malicious extension is the way chosen for the attack (like in our main case study), then it is required to fool the victim on the client side by finding a way to surreptitiously install such extension, for example, with some social engineering trick in order to persuade the victim to do so. In the case of an XSS exploit, on the other hand, a vulnerability should be found in

the first place, and this, in many cases may prove neither easy nor likely to happen. However in presence of a successful XSS exploit, it would be more difficult for the victim on the client side to realize that he/she is under such kind of attack or even to prevent it, since no additional operation by the victim is required such as malware extension unconscious installation.

5 Software architecture: design choices and implementation

The Scenario 3 just discussed served to present the BitM + attack idea aimed at defeat- ing FC2W strong authentication. We now proceed to describe the software technologies and architectural design choices that led us to implement a working BitM + attack prototype. Basically we are illustrating a weaponizable Proof of Concept (PoC), the components involved in it and how they cooperate in order to conduct to a successful phishing attack against FC2W.

The entire BitM + architecture is divided into two main parts communicating with each other:

Back-End (BE): Implemented with an enhanced BitM design, the Back-End communicates with the Front-End via HTTP/WebSocket. It represents actor B in Scenario 3.

Front-End (FE): JavaScript code injected through an XSS vulnerability (xss-Payload), implementing MitB. It operates within the victim's (actor V's) browser in Scenario 3.

A set of software utilities, environments and frameworks have been involved into the development of both BE and FE.

5.1 Involved software tools in BE

5.1.1 Node.js

Node.js consists in a versatile, open-source server environment that functions seam- lessly across multiple platforms [40–42], including Linux, macOS, Windows, and more. As a back-end JavaScript runtime environment operating on the V8 JavaScript engine, Node.js executes JavaScript code outside the confines of a web browser. Node.js covers a significant part of our BE implementation, in fact many of the libraries and scripts coded in the BE—that serve for the attack—work under Node.js.

Node.js is usually leveraged to employ JavaScript for creating command line tools and executing server-side scripting. The capability to run JavaScript on the server is commonly utilized to dynamically generate web page content before delivering it to the user's browser. Consequently, Node.js embodies a "JavaScript everywhere" paradigm, streamlining web application development by using a single programming language for both server and client-side programming, as opposed to employing different languages. Such set of characteristics makes Node.js a very useful and flexible environment for the attack approach proposed here, also making it sufficient for the attacker to write the required code by using just a single programming language (JavaScript) for both.

the BE and FE sides of the attack architecture.

5.1.2 Puppeteer API

Puppeteer is a Node.js library which provides a well-documented and maintained [43] high-level API to control Chrome/Chromium browser over the DevTools Protocol. For our attack implementation needs, the interesting point is that almost every task normally achievable in the browser by manual user operations can be programmatically accomplished through Puppeteer library, that provides the attacker with an extremely powerful and convenient tool to work with. In other words the programmer has full control of the Chromium browser instance launched and handled by Puppeteer and is able to automate every aspect of its behaviour and functionalities (from the most external feature of the native browser application itself to the most restricted and sandboxed context such as the DOM content of an HTML page opened in it) by just using JavaScript code. Remember that all of this happens in Node.js environment that can also directly provide OS level APIs such as, for example, disk file I/O and a lot more. All in all, Puppeteer offers a versatile range of capabilities, allowing users (or even attackers) to automate various tasks such as form submission, UI testing, and keyboard input with the ability to create automated testing environments. Puppeteer can be also used for efficient web pages and Chrome extensions testing providing a comprehensive toolkit for web development and testing needs.

5.1.3 Chromium browser

Chromium is a web browser project that is both free and open-source, is primarily developed and maintained by Google [15, 44]. This codebase serves also as the founda- tion for the Google Chrome browser, which, in contrast, is proprietary software with additional features.

Widely utilized, the Chromium project is the basis for various browsers, including Microsoft Edge, Samsung Internet, Opera, Vivaldi, and several others. Additionally, numerous app frameworks incorporate significant portions of the Chromium code.

In our design a release of Chromium has been chosen to work with Puppeteer API.

5.1.4 Express.js

Express.js is a backend web application framework for developing RESTful APIs using Node.js. It is designed for the development of web applications and APIs [45], it represents the de facto standard server framework for Node.js [46].

It is simple and minimalist and that makes it the most obvious choice for the implementation of a malicious API server in attacker's BE. The introduction of this component in the BitM + design represent the main addition in terms of set of libraries and tools used in BE side with respect to BitM presented in [9].

5.1.5 VNC/noVNC

Virtual Network Computing (VNC) is a technology that enables remote access and control of a computer or server over a network. It allows a user to view and interact with the desktop environment of a remote system as if they were physically present in front of that machine. VNC operates by transmitting the graphical desktop data from the remote computer to the viewer application, and it sends back user inputs, such as mouse and keyboard actions. It also includes clipboard copy/paste features with full Unicode support. This technology is widely used for remote administration, support, and collaborative work, providing a means to access and manage computers from a distance. Such VNC functionalities are typically provided by using a VNC server (native application installed and running on the remote machine that needs to be controlled) and a VNC client (native application installed and running on the machine used by the user to control the remote one). In this way the remote machine with VNC server running exposes the desktop environment on the network so the user with the VNC client can connect to it.

noVNC is both a HTML VNC client JavaScript library and an application built on top of that library [47].

Under the hood noVNC can be seen as basically a web server (typically imple- mented with nginx [48]) that uses WebSockets [49] and HTML canvas to deliver reliable and responsive real-time VNC session (at the base of the BitM/BitM + working prin- ciple) just via browser using just web technologies and hence dismissing the necessity of an ad-hoc native VNC client application installed in the user's (or victim's) OS.

The noVNC WebSocket server runs—in our BitM + design—on BE machine. BE runs also a native VNC server app. The purpose of noVNC WebSocket server is to gather VNC session provided by the native VNC server app and deliver it via HTTP/WebSocket on an HTML canvas in a webpage exposed by the noVNC library itself and made available to web browser. So noVNC can be seen as a VNC session proxy that takes the VNC session from the VNC server running on its own machine and forwards it via WebSocket connection. At this point, noVNC server is also responsible at exposing a web application (a simple webpage)—navigable on FE by user's/victim's browser—that contains an HTML canvas that displays the VNC remote desktop envi- ronment. In the case of our attack, the VNC remote desktop environment shows the victim the BitM + Chromium window opened and running, giving him/her the illusion that the page he/she is navigating is being navigated on his/her local browser, while it is actually navigated on the BitM + Chromium browser running on BE attacker's machine.

The just described mechanism is also visible in Fig. 10 which shows the transmis- sion of BitM's browser window page from the attacker's machine to victim's browser with a remote desktop protocol such as, in this case, VNC handled by the noVNC library.

noVNC uses many modern web technologies so a formal requirement list for the browser is difficult to produce. However the list of the minimum versions of the cur- rently known compatible browsers includes: Chrome 64, Firefox 79, Safari 13.4, Opera 51, Edge 79. So all these browsers and their later versions support (and hence can potentially deliver) a BitM/BitM + attack to the user.

To sum up, the main steps to set up a noVNC working session are: run a VNC server on the BE side and run a WebSocket proxy that points to the VNC server, and then, on FE side, load the page and connect.

Note that VNC protocol and the noVNC project [47] are an open source alterna- tive to, for example, Microsoft's RDP proprietary protocol. It can be shown that RDP can also be used as a replacement for VNC/noVNC component in the BitM + imple- mentation, in fact even if RDP is a proprietary protocol, some of its specifications are available and other parties have developed their own RDP clients. Many examples of independent web based (HTML5/JS) implementations of RDP clients are available online on GitHub. However noVNC arguably represents the best and most balanced choice to date for BitM/BitM + attack in terms of support, source code availability and features reliability.

5.1.6 ngrok

Ngrok is a tunneling and reverse proxy service that allows to expose a local server to the Internet via HTTPS protocol. It provides a secure way to share and test local web development projects or APIs with others.

ngrok is usually used during the development and testing phases of web applica- tions, especially when dealing with webhooks, APIs, or any scenario where external services need to connect to a local server. So even if ngrok is primarily designed for development purposes—since it can

expose over HTTPS, with little configuration, a web application on the Internet from a local machine server—it has been chosen as component of BitM + attack responsible for delivering the malicious attack's webpage and its APIs.

Technically it is a service that can translate a server which is running over insecure HTTP localhost (that can be potentially reachable only from the server itself and the other machines in the same LAN) to a secure HTTPS server with a static address reachable from anywhere on the Internet.

From a BitM + attack's technical perspective, one of the most important ngrok's features is that the web resources are provided over HTTPS protocol, which is a mandatory requirement in the specifications of WebAuthn APIs: such APIs' methods (e.g.: navigator.credentials.get()) can be called only from an HTTPS-secured webpage context.

5.1.7 Docker

Docker is a platform-as-a-service (PaaS) tool, utilizing OS-level virtualization, designed for the creation, deployment, and execution of applications through con- tainerization. Containers are self-sufficient packages that encapsulate all the necessary components for running software, including code, runtime libraries and system tools. The core of Docker is the Docker Engine, comprising a server, REST API and command-line interface for interacting with Docker. Docker images, lightweight and standalone packages, serve as the blueprints for containers. These images can be stored in repositories like Docker Hub, with Dockerfiles providing the instructions for image creation. Containers, instances of Docker images, offer portability and consis- tency across different environments, aiding in reliable application execution. Docker simplifies development and deployment by ensuring standardized and reproducible application packaging and execution, leading to increased efficiency and consistency in software workflows.

So the key principle of the Docker tool workflow here is that the Dockerfile repre- sents the "recipe" containing all the instructions that determine the build of a virtual environment's image. An image can be seen as a static snapshot of the state of that virtual environment. From that "frozen" snapshot which is the image, a container instance of it can be run. In fact, the container executes and "brings to life" the actual virtual environment contained in the image. Then, where the image is a self-contained static standalone file, the container launched out of it is the running instance of the virtual environment contained in the image along with all the host system (the OS hypervisor) resources allocated in order to run it.

All these features make Docker a very suitable tool for BE side BitM + attack development. As it not only allows for a consistent and compact architectural design of the BE implementation, but also allows for good scalability and portability in a future perspective.

In particular for our PoC, an already available boilerplate repository on GitHub has been used [50, 51]. Such repository consists of a Docker image that runs a container with a basic configuration that offers noVNC connectivity between the container and the host OS. Then on adopted, such repository has been further developed in order to insert and code the components needed for the BE implementation of the BitM+ . Similarly other equivalent boilerplate Docker repositories or a Docker image written from scratch can be used to achieve the same results. Moreover the BitM + attack can prove just as effective also if implemented without using Docker containerization at all: the required software code can be written, configured, and deployed directly on the attacker's machine host OS (both Linux and Windows based OSes were tested). This will not impact or alter the effectiveness of the attack. However a Docker con- tainerization of the attack's BE appears to be a better design choice from a purely technical point of view.

5.2 Involved software in FE

5.2.1 Browser (client-side) vulnerability exploitation: JavaScript code injection through Cross-Site Scripting (XSS)

Cross-Site Scripting (XSS) is a type of security vulnerability commonly found in web applications. It occurs when an attacker injects malicious scripts into web pages viewed by other users. These scripts can execute arbitrary code in the context of the vic- tim's browser, allowing the attacker to steal sensitive information, manipulate the appearance of the web page, or perform actions on behalf of the user.

Reflected XSS is one of the types of XSS attacks and the one used in this work to perpetrate the presented attack. In particular, in a reflected XSS attack, the malicious script is injected into a web application and then reflected back to the user's browser within the response from the server. This often happens when user input is not properly validated or sanitized by the server before being included in the server's response.

5.3 Back-end side tasks

The software that implements the BE resides in the attacker's machine. Such BE machine has been tested to work either with Windows or Linux host OS. Also, a fully functional cross-platform containerized version of the BE has been developed with Docker which is the version that will be described here. BE's main tasks are:

1. BE-Task1: Implementing the improved BitM by using a Puppeteer-controlled Chromium browser instance (BitM at its core is nothing but a regular web browser with specific software modifications aimed to programmatically modify its behaviour);
2. BE-Task2: Providing remote BE I/O functionality—i.e. BitM webpages' live video streaming coupled with mouse and keyboard input capability—to the FE through Internet;
3. BE-Task3: Providing ad-hoc HTTP APIs over the Internet for WebAuthn API- related data exchange between BE-FE and other optional features.

5.4 Front-end side tasks

The FE is implemented with a JS script served by the attacker B and injected via Reflected XSS into RP page (we simply call it xssPayload). It represents the MitB component of the attack. Note that use of an XSS exploit as vector of attack works on practically any browser including Google Chrome, Microsoft Edge, Mozilla Firefox or Apple Safari. FE main tasks are:

1. FE-Task1: Overlaying BitM webpage remote session on a target page that meets specific conditions in its URL (we call such target xURL);
2. FE-Task2: Periodically checking for new challenges from BitM (originally pro- vided by RP) by using HTTP polling;
3. FE-Task3: Receiving the challenge and then call WebAuthn API's get() method on victim's browser in order to generate the assertion with victim's HW authenticator and sending it back to BE.

5.5 Back-end side components

Now that a complete background on technologies involved in the presented BitM + attack as just been provided, we will proceed to describe the very attack implemen- tation—which relies on such technologies—that has been carried on both BE and FE parts. Let's start from the BE: we now describe the single components that implement it and try to explain how such components cooperate in order to deliver a successful BitM attack. The BitM + attack components specifically written by the attacker in the BE are:

1. mybrowser.js Node.js main script file;
2. bitm overrider pup.js JS injection script file;
3. myutils.js utilities script file used by bitm overrider pup.js;

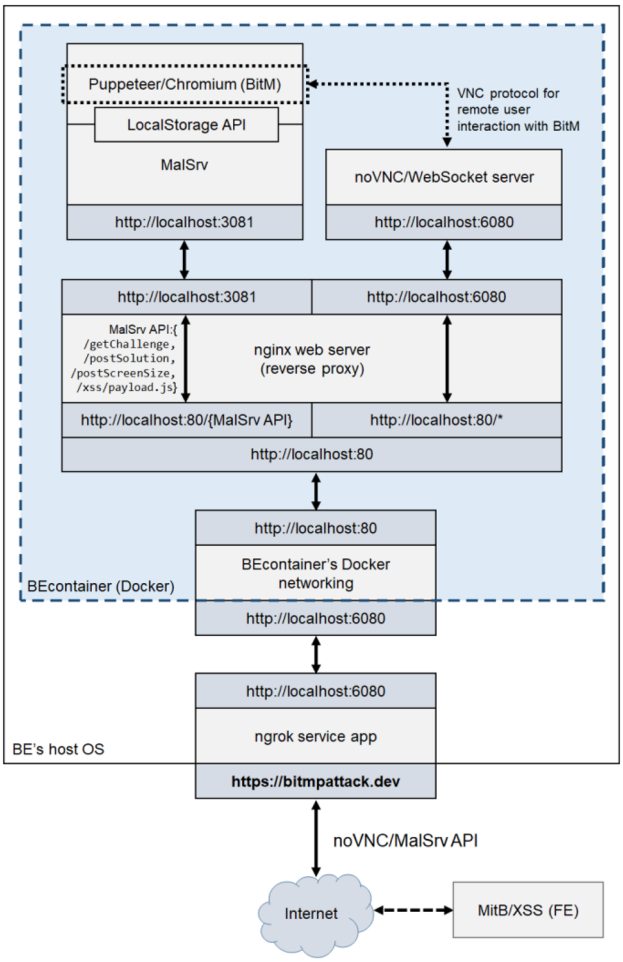

Fig. 15 Block diagram of the BE's network ports configuration and high level overview of the BE architecture

4. noVNC.html web application entry page for BitM + attack delivery via noVNC.
5. payload.js JS script file served by BE in order to be injected into RP's XSS- vulnerable webpage.

We will now examine each one of them. All these the components are illustrated in the BE block portrayed in the top area of Fig. 15.

We assume that each one of the components listed above is located inside a Docker image (BEimage) which provides the containerized Docker environment (BEcontainer) that runs the BE side of the BitM + attack.

5.6 mybrowser.js

This script file runs under Node.js which is installed (and runs) in BEimage docker con- tainer (as shown in Fig. 15). It represents the core of BitM functionality: it imports Puppeteer API modules that launch a fully programmatically controlled Chromium browser instance. Such

Chromium instance implements the BitM itself and—under the typical attack circumstances—it navigates the legit authentication web page of a RP (RPpage) service account. mybrowser.js—through Puppeteer—is also responsi- ble of the injection of the JavaScript code into the RPpage context. The JS code to be injected into RPpage is contained in bitm overrider pup.js, and it will be described later. Inside mybrowser.js it is also implemented a web server by using Express.js library. It provides REST APIs that serve different tasks for BitM + to work. As it can be noticed in Fig. 15 such web server (we call it *MalSrv*) exposes a set of REST APIs (we call them *MalAPI* s). These API work together with built-in Chromium localStor- age APIs [52] that are used to keep track of chObj and assertObj involved during the attack process. In order to do this two varibles are instanced: "mychallenge" and "mysolution", the first one stores the chObj received by the legit RP (Relying Party) waiting to be transmitted to the FE's MitB/xssPayload and the latter stores assertObj received back from FE's MitB/xssPayload. Both variables are initially empty and set to *null* by default. They can be seen as some sort of temporary "recipients" that enable communication between the Chromium browser instance itself (representing the BitM component) and the MalSrv component, both implemented inside the mybrowser.js script. Now we proceed to list and further detail the MalSrv APIs, which are:

1. *GET* /getChallenge: it has no parameters and it is periodically called (polling) by MitB (xssPayload) in the FE and responds with a new chObj if any is available. The chObj is gathered from a localStorage variable—named "mychallenge"—stored in Chromium browser instance. Such variable is updated and filled with a new chObj whenever a login request by the victim has just been remotely performed from FE—via noVNC—on the BitM.
2. *POST* /postResult: it is called by MitB (xssPayload) in the FE. The payload in its body contains a valid assertion JSON object (assertObj) that is sent to MalSrv in BE. When this API is called, the assertObj is saved in "mysolution" localStorage variable that was previously emptied (set to *null*).
3. *POST* /postScreenSize: it is called by MitB (xssPayload) in the FE every time a JS window.onresize event is fired in the BitM + page opened on the victim's browser. This POST request's body is a JSON object formatted as "{w: xxxx, h: xxxx}" (e.g.: {w: 1704, h: 692}) which tells the BE the new size (height and width values) of the victim's browser window in order to resize the correspondent BitM + Chromium window in BE accordingly (and hence transmit back to the FE the BitM + 's web page view with the correct updated size). This ensures a better and seamless victim's browsing experience by improving responsiveness of BitM + attack's page, thus making more difficult for the victim to spot the attack.
4. *GET* /xss/payload.js: it serves the script file payload.js which represents the XSS code payload (xssPayload).

Such MalSrv APIs are exposed on localhost's port 3081 (http://localhost:3081) in Docker container's virtual environment.

5.7 bitm overrider pup.js

Contains the JavaScript code that needs to be executed inside the RPpage. When RPpage is navigated, the script browser.js—through Puppetter API—injects the bitm overrider pup.js code in RPpage. The specific bitm overrider pup.js con- tains the code that overrides the original navigator.credentials.get() method in RPpage with a new function. When we are in presence of an attack situation, such new function we named evilGet() (already introduced in Scenario 3) saves the chObj—provided by RP to RPpage—into the "mychallenge" localStorage variable, so it can be made available from MitB/xssPayload in FE by polling the MalSrv's.

/getChallenge API which basically reads the "mychallenge" content (which can be either a chObj or a *null* value if no challenge is available at the time the API is called) and sends it as response to the FE. In the meanwhile the evilGet() call does not return and stays alive waiting for a new assertObj by checking the content of the "mysolution" localStorage variable until it is not *null*. A few moments later, when the FE, in possession of chObj, manages to extract the assertObj from the victim, then the FE calls /postSolution POST API with assertObj in its body. /postSolution once called on the MalSrv receives and saves the assertObj in the "mysolution" local- Storage variable. Now the "mysolution" localStorage variable is not *null* anymore as it contains the required assertObj. So the evilGet() function can finally return assertObj to RP server in order to complete the authentication. After the authen- tication completion both "mychallenge" and "mysolution" variables are flushed and set back to *null*. The above described mechanism is the main task performed by the bitm overrider pup.js script and it is also illustrated in a simplified manner in Fig. 13.

5.8 myutils.js

To perform its tasks, the JavaScript code contained in bitm overrider pup.js imports some auxiliary functions provided by myutils.js. Such code contains function that perform serialization/deserialization of data of chObj and assertObj. This is required since both chObj and assertObj—that are generated, exchanged and used by RP (Relying Party) and HW authenticator during the FC2W process—contain JSON data with ArrayBuffer [53] data type. Data of such type cannot

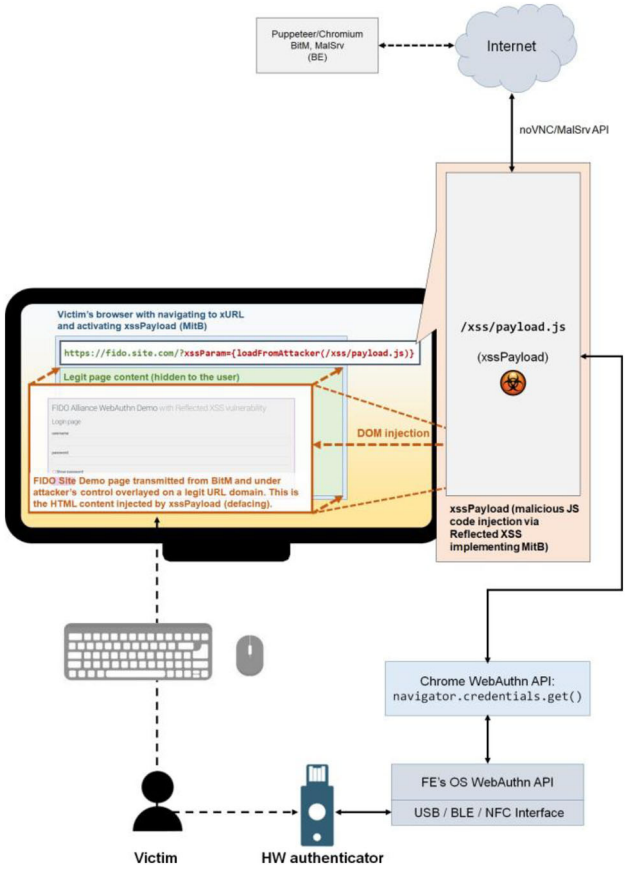

Fig. 16 Block diagram of the FE's xssPayload/MitB implementation

be transmitted via REST API right away. First, data need to be serialized and encoded in form of stringified JSON fields in order to be suitable for transmission via MalSrv HTTP APIs, and then converted back in ArrayBuffer type when they need to be evaluated by the two end points: the navigator.credential.get() method in FE must take as an argument a chObj in ArrayBuffer data type. Similarly the assertObj must be in ArrayData buffer format when it is returned by evilGet() to the RP server. The diagram in Fig. 16 highlights such data conversion mechanism.

5.8.1 noVNC.html

The noVNC.html code presents to the victim the BitM + Chromium browser RPpage making him/her believe he/she is in front of a legit page directly provided by RP (Relying Party) server. Such page in the present implementation is exposed by an nginx web server pre-installed and configured inside the Docker BEcontainer made available at [51]. Such nginx server—listening on port 80—also serves as a reverse proxy and we have modified its default configuration file inside sites-available folder in order to redirect to MalSrv—active on port 3081 from mybrowser.js (still inside the BEcontainer)—exclusively the HTTP API's traffic related to it (i.e.: /getChallenge,

/postSolution, /postScreenSize). This means that any request pertinent to Mal- Srv arriving on port 80 from outside the BEcontainer is forwarded/proxied to port 3081 (the MalSrv's port). The rest of the HTTP/WebSocket traffic that arrives on port 80 is handled by nginx server and redirected to the noVNC server on port 6080 in order to deliver the typical VNC connection functionality on which the BitM + relies. noVNC.html page represents the page BitM + attack entry point and it is exposed by nginx inside the Docker *BEcontainer* at http://localhost:80 and binded and made accessible to the FE's host OS machine on http://localhost:6080. So the Docker container, in this case, publishes to the host OS the port 6080 when running.

5.8.2 payload.js

It is requested during the Reflected XSS RP (Relying Party) website exploitation when xURL is navigated by the victim. It contains the xssPayload and represents the MitB implementation itself on FE. Once requested, the code content is injected into the RP's XSS-vulnerable webpage.

5.9 ngrok service: from localhost page to HTTPS page on the Internet

The ngrok service app is configured and run on FE's host OS (hence outside BEcontainer) in order to expose to the Internet the local noVNC.html page that car- ries the BitM + attack, which otherwise would be only locally available—as mentioned.

- at http://localhost:6080. So ngork translates http://localhost:6080 to a public Inter- net HTTPS domain (e.g.: https://bitmpattack.dev on the default 433 HTTPS port). At this point, the entry point web page (i.e. noVNC.html that we will call *ePage*) of our BitM + attack is made available potentially to anyone on the Internet. The BE's network ports configuration and overall architecture is described in the big picture of Fig. 15, however Fig. 17 offers a more abstract and essential representation of the BE that helps to better explain how the different BE's components are organized and interconnected.

5.10 Front-end side components

We are now describing the relevant features on the FE side i.e.: the xssPayload—which implements MitB attack on the victim's browser (Figs. 18, 19, 20, 21, 22, 23, 24, 25 and 26).

We will explain how such component works with the BE in order to deliver a successful BitM + attack. The features of xssPayload which basically is a JS script file served by

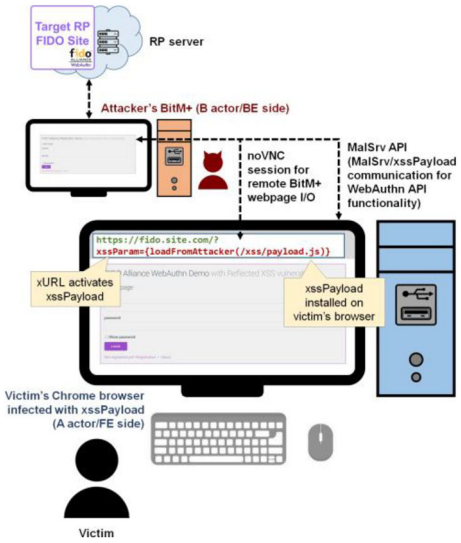

Fig. 17 BitM + attack: stage 1. Here the actors and the starting conditions of the attack scenario are described

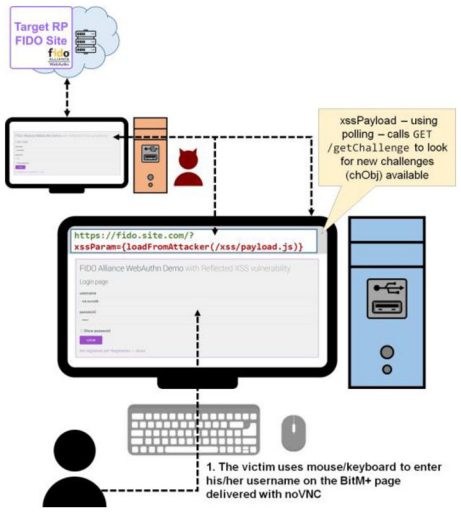

Fig. 18 BitM + attack: stage 2. The victim uses mouse and keyboard devices to insert his/her username on the login page transmitted from BE. Until this point BitM + attack has no difference with regular BitM in terms of functionality

the attacker server (MalSrv) and injected inside the XSS-vulnerable RP's webpage are listed as follows:

1. arbitrary code injection into target page (xURL page).
2. communication tasks via HTTP REST API with attacker's BE, in particular MalSrv.
3. defacing the HTML DOM content of the RP's vulnerable webpage: it overlays malicious HTML content on original RP page content (reached by opening xURL) in order to provide BitM webview streaming via noVNC.

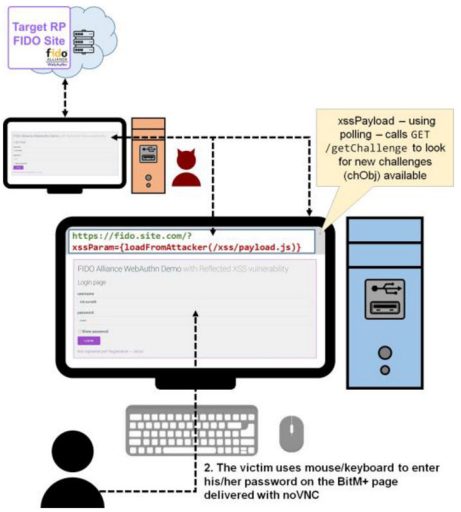

Fig. 19 BitM + attack: stage 3. The victim uses mouse and keyboard devices to insert his/her password on the login page transmitted from BE. Until this point BitM + attacks has no difference with BitM in terms of functionality. From now on BitM + innovations are used in order to carry on the attack against FC2W

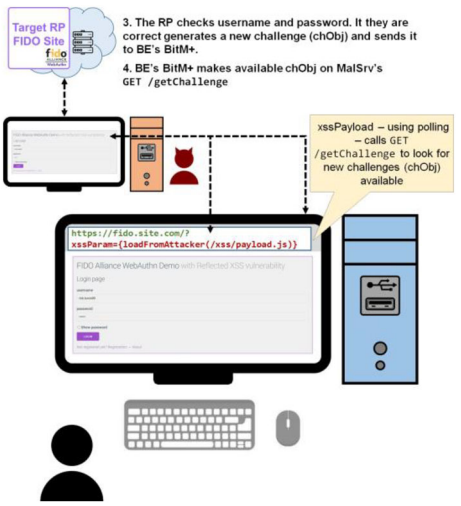

Fig. 20 BitM + attack: stage 4. The BitM sends to RP (FIDO Site Demo server) the credentials remotely inserted by the victim from FE. The RP validates the credentials, if they are valid it generates a new challenge (a new chObj) that is provided to BitM + running on BE. At this point chObj is passed to MalSrv which publishes it by making it available through the /getChallenge API

The FE side and its components discussed in the following are portrayed in Figure.

16.

5.10.1 Code injection

Code injection capability grants the attacker the main and most important advan- tage of implementing such MitB attack, i.e. the possibility to arbitrarily inject any JavaScript

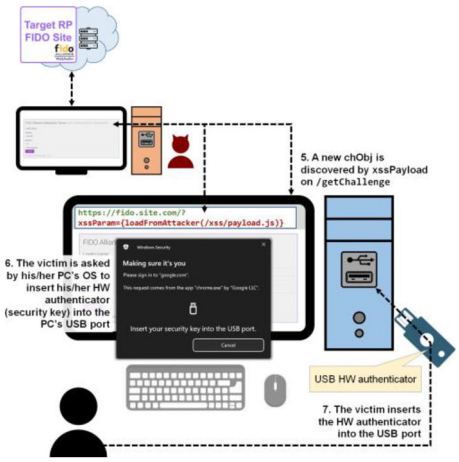

Fig. 21 BitM + attack: stage 5. The xssPayload, which in the meantime was polling MalSrv on /getChallenge, discovers the new published chObj. Such chObj is passed to the original WebAuthn API navigator.credentials.get(chObj) method, which is called inside the xURL page's context. Such context is trusted since it exhibits the legit'fido.site.demo' domain. So the browser calls in question the victim's host OS which uses its native WebAuthn API to spawn a system authentication window that asks the victim to insert his/her HW authenticator. Then the victim inserts his/her HW authenticator in his/her PC's USB port as requested

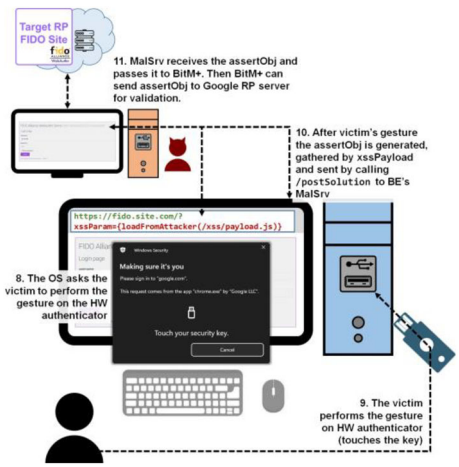

Fig. 22 BitM + attack: stage 6. Once the victim inserted the HW authenticator in the USB port of is machine (the FE machine), the confirmation window ask the victim to perform the gesture (e.g. touch the key) to confirm authentication. After victim's gesture the assertObj is generated, gathered by xssPayload and sent to BE's MalSrv by calling /postSolution. MalSrv receives the assertObj and passes it to BitM + . Then BitM + can send assertObj to FIDO Site Demo RP server for validation

code into a webpage navigated by the victim's and directly pro- vided by RP server (for example the vulnerable XSS demo site reachable at https://fido.site.demo). xssPayload is injected when the victim navigates the spe- cific target page xURL that can be exemplified as https://fido.site.demo/*?xssParam = loadFromAttacker(/xss/payload.js). Note that such xURL exhibits a legit target RP (Relying Party) domain

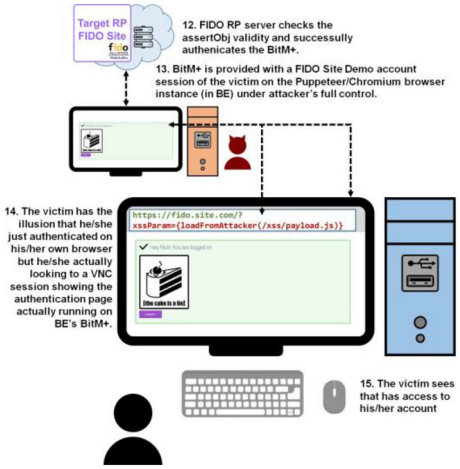

Fig. 23 BitM + attack: stage 7. The RP server (FIDO Site Demo) checks the assertObj validity and successfully authenticates the BitM + . BitM + is provided with a FIDO Site Demo Account session of the victim on the Puppeteer/Chromium browser instance (in BE) under full attacker's control. The victim has the illusion that he/she just authenticated on his/her own browser but he/she actually looking to a VNC session showing the authentication page actually running on BE's BitM + . The victim press "Next" to enter to continue and enter the account

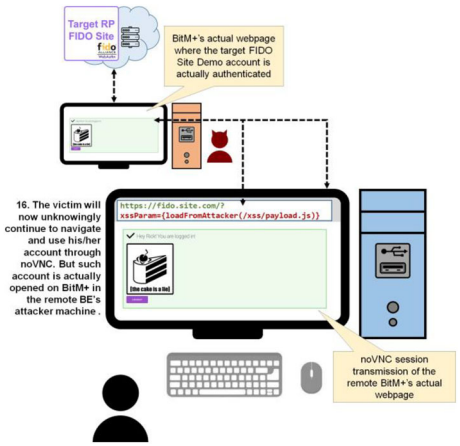

Fig. 24 BitM + attack: stage 8. From now on, the victim will now unknowingly continue to navigate and use his/her account through noVNC. But such account is actually opened on BitM + in the remote BE's attacker machine

with a parameter (xssParam) responsible for the XSS vulnerability and used as the client-side attack entry point. xssPayload proceeds to alter the original xURL page and overlays, on the xURL page's DOM document (that is so far exhibiting the legit fido.site.demo login page), an iframe element that covers the entire xURL page's webview (see Fig. 10). Such iframe ele- ment has the src attribute set to https://bitmpattack.dev and hence it will load and present to the victim the *ePage* (which essentially is the noVNC.html served by BE and exposed on the Internet with ngrok). *ePage* will present the remote Pup- petter/Chromium BitM instance

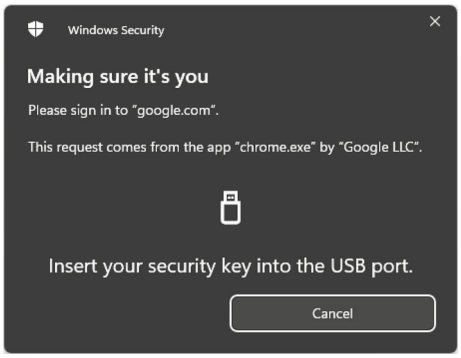

Fig. 25 Security window spawned by native Platform WebAuthn API on the victim's host OS (in this case Windows 11)

showing a legit fido.site.demo (FIDO Site Demo RP page) login page. So the victim is induced to interact and insert via VNC his/her credentials in the remote FIDO Site Demo RP login page controlled by BE's Pup- petter/Chromium BitM instance. However, what has been described so far in this paragraph is part of the normal BitM approach.

xssPayload is also responsible of calling the regular navigator.credentials.get() (or simply get()) method when a chObj—origi- nally generated from RP (Relying Party) in BE is discovered during polling with.

/getChallenge. The chObj contains the rpId field set to 'fido.site.demo'. So the get() method can be successfully called only in pages having fido.site.demo as URL (see the SC1 security check described in Scenario 2 and 3) which is the case. Once get() method has been called and the browser through the victim's OS has obtained the assertObj from the HW authenticator, xssPaylaod is committed to send assertObj to B's MalSrv via POST /postSolution. xssPayload is also responsible to listen at window.onresize events fired by the xURL page and to send the updated page's size data (height and width) to B's MalSrv via POST /postScreenSize. xssPayload also includes the equivalent code contained in myutils.js in order to correctly perform the serialization/deserialization opereation required when dealing with chObj and assertObj (see Fig. 16).

5.10.2 Communication tasks

xssPayload—as previously mentioned—communicates with BE via the HTTP APIs provided by MalSrv. One of its main tasks is to look for new chObj from BE's MalSrv by periodically polling it with API. When a new chObj is discovered it is passed WebAuthn API's get() method. It also does the complementary task, i.e. it takes the assertObj as result of the get() call and sends it to BE's MalSrv by using/postSolution with assertObj in the POST's body.

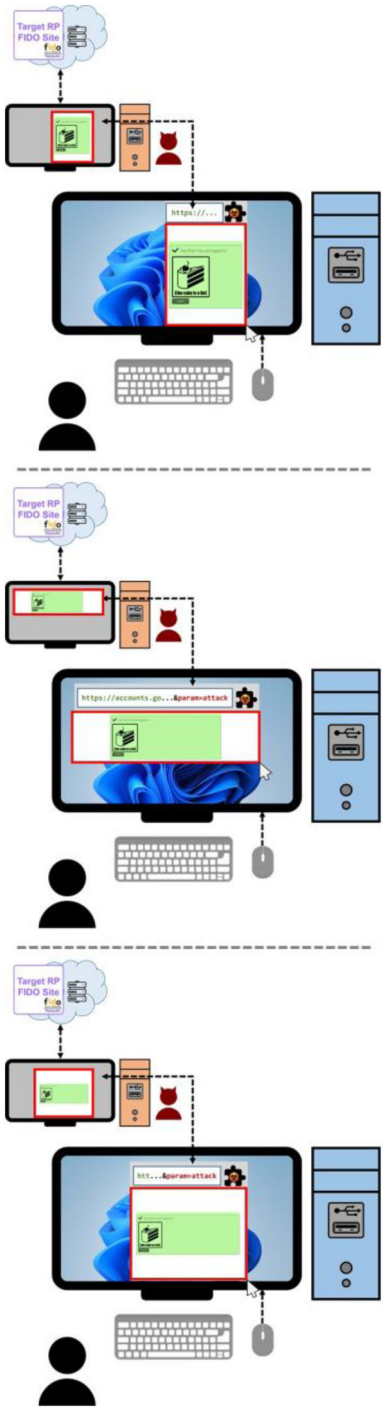

Fig. 26 Three examples of BitM + window resize responsiveness. The BitM + is aware of the window resize produced by the victim with the mouse on his/her local infected browser and updates in real time the window size of noVNC transmitted webpage accordingly. The data for window resize update are sent by xssPayload to MalSrv via /postScreenSize POST request. The red-bordered areas highlight the window size synchronization between the BitM + in BE and the victim's browser in FE

5.10.3 DOM defacing

xssPayload performs defacing by injecting HTML code which contains the DOM ele- ment (an iframe that loads the ePage) to be overlaid (see Fig. 10) over xURL page's original and legit content that—in this case—is the RP's (FIDO Site Demo) login web page.

6 Real-world attack experiment

After describing the practical software implementation of the BitM + by showing its architecture, the involved technologies and the design choices, we now proceed to show our presented BitM + at work in a real-world scenario, in order to demonstrate that the proposed new BitM + is not only theoretically conceivable but also practically feasible today. We will illustrate the various steps involved in a real attack use case that lead to a successful authentication of the attacker to a target FC2W-protected victim's account. We assume the following set-up (compatible with Scenario 3 conditions) immediately before the beginning of the attack:

1. V (the FE): the victim uses Chrome browser that will navigate an XSS-vulnerable RP webpage (reachable at xURL). Such page will execute xssPayload that implements a MitB attack.
2. B (the BE): Attacker's BE machine implements and runs controlled Pup- peteer/Chromium browser instance with MalSrv APIs and noVNC connection functionalities (BitM +).
3. RP: the Relying Party server implements FC2W. In this particular experiment the FIDO Site Demo login page (https://fido.site.demo) is targeted.

The versions of BE's software components used for the attack are hereby listed:

- Node.js v18.16.1
- noVNC v1.4.0 from GitHub repository [47].

The Node.js dependencies used by mybrowser.js in the BE are:

- express: v4.18.2
- puppeteer: v21.3.8
- puppeteer-extra: v3.3.6
- puppeteer-extra-plugin-stealth: v2.11.2

Puppeteer API uses and controls Chromium v120.0.6055.0 (× 64) browser.

6.1 BitM + at work

Once the above configuration is in place we are ready to start the BitM + attack against the victim in V. We go through the process with the help of figures representing the different stages of the attack. Figure 19 (stage 1) offers an overview of the situation at the beginning of the attack. The following Figures in this subsection, from 18 to 24, describe the sequence of all the stages that take place until the successful completion of the attack.

6.2 Experiment observations

With stage 8, the BitM + attack procedure is complete, resulting in a full attacker's control of the victim's FIDO Site Demo Accounts session that resides in BE's Chromium browser and is only transmitted via VNC to the victim in FE.

Let's note the following:

6.2.1 Native Platform WebAuthn API

The security window showed in Fig. 14 (and also appearing in Figs. 23 and 24) is originated—upon victim's infected browser request—directly by the host OS in the victim's machine (V) and it is not part of the transmission of the BitM + webview via noVNC. In fact such window in Fig. 14 is triggered by xssPayload in victim's browser that calls the WebAuthn API method navigator.credentials.get(). The victim's browser in turn forwards the call to the OS that is responsible for displaying the security window.

Moreover in the security window in Fig. 14 it is also readable the Relying Party, that is in that figure example is "google.com" but it can be done with the domain "fido.site.demo" or any target RP (Relying Party) domain which exhibits XSS attack exploitability. Such information comes from the string value in rpId field contained in chObj along with the challenge and other optional data specified by the WebAuthn API standard. This means that rpId provided in chObj must match the one stored inside the used victim's HW authenticator at the moment of registration (SC2).

We can also point out that the host OS believes that "This [authentication] request comes from the app'chrome.exe' by'Google LLC'." as it can be read from the window itself in Fig. 14. The referred "chrome.exe" is the infected victim's browser that is considered as trusted. So the host OS does not consider the remote BitM + (which is running on a Puppeteer/Chromium instance, on the attacker's BE machine potentially inside a containerized Linux environment) as the caller of the authentication but the local victim's browser itself (in this case Google Chrome running under Windows 11) infected with xssPayload. This means that hardly any difference can be spotted by the end user between a regular FC2W user authentication and one under a BitM + attack,

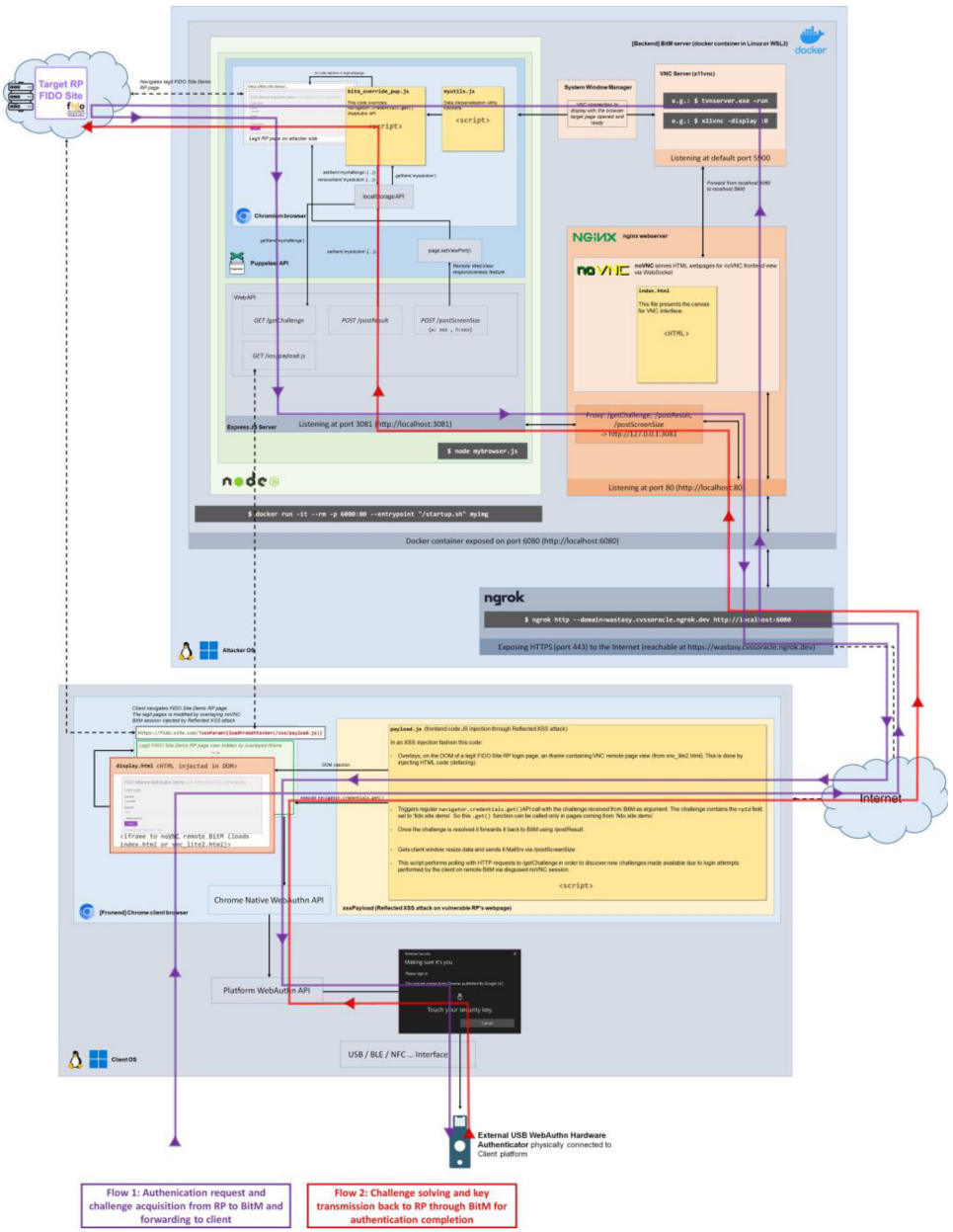

Fig. 27 Overall BE/FE software architecture and attack procedure flow. Flow 1 path begins with victim's user credentials insertion on the BitM page and ends with chObj arrival to HW authenticator. Flow 2 path begins from chObj arrival to HW authenticator and the generation of assertObj and the terminates with the arrival of assertObj to RP server which determines a successful authentication on BE's BitM

since from a graphical user experience standpoint they are indistinguishable: the security window generated by a regular FC2W authentication is exactly the same as the one generated under a BitM + attack and showed in Fig. 14.

6.2.2 BitM + webview responsiveness

As already mentioned BitM + implements webview resize responsiveness which enable real-time adaptation of the page DOM content according to the potential window resize by the user/victim. A visual representation of such feature is illustrated in Fig. 27.

7 Future work

Despite the achievements of the proposed new attack, which stands as a further improvement of the basic BitM idea covered in [9], there is room for further research and improvements on phishing attacks. For example, the technology behind remote transmission of Web pages, currently implemented with noVNC, might be probed to promote the development of an even more compelling user experience with lower latency and response time to input. However, it is worth noting that, at the present time, BitM + attacks also defeats different kinds of anti-phishing AI-based approaches like the one proposed by C. Catalano et al. [54].

8 Conclusions

The research presented so far sheds light on the evolving landscape of web secu- rity, emphasizing the vulnerabilities associated with the widely adopted Multi-Factor Authentication (MFA) methods, specifically in the context of Browser-in-the-Middle (BitM) attacks. While the FIDO2 Project, incorporating the CTAP2 protocol and Web Authentication API (WebAuthn API), has stood resilient against contemporary BitM implementations, this study advances the field by proposing an enhanced BitM attack pattern.

Our work extends the applicability of BitM attacks by introducing a novel technical architecture that combines BitM with Man-in-the-Browser (MitB) techniques. This innovative approach enables the compromise of any available MFA method, includ- ing FIDO2/WebAuthn solutions that rely on hardware dongles—an authentication method previously undefeated by phishing attacks. The proposed attack is designed to exploit vulnerabilities in web browsing security, facilitated either through a malicious browser extension or malware installed on the client's operating system.

A noteworthy aspect of our research is its focus on the communication between the Relying Party (the server providing web service account access) and the client's computer. Rather than targeting the cryptography technology of FIDO hardware authenticators, our BitM + attack operates on the WebAuthn APIs, which have become a standard in modern browsers for enabling strong authentication. By focusing in on the navigator.credentials.get() JavaScript method within Web Authentica- tion API, our attack demonstrates a capacity to bypass even the most robust security protocols.

It is important to note that this work builds upon prior research [9, 10], offer- ing an evolved and more potent version of the BitM attack. While not undermining the inherent security of FIDO hardware authenticators, the proposed BitM + attack exposes vulnerabilities in the communication channels crucial for the authentication process. As web authentication technologies continue to advance, the results here presented underscore the need for constant vigilance and iterative improvements to counter emerging threats in the digital realm.

Acknowledgements This work was partially supported by the following project: SERICS—"Security and Rights In the CyberSpace—SERICS" (PE00000014) under the MUR National Recovery and Resilience Plan funded by the European Union—NextGenerationEU.

Author Contribution The research idea, the modeling of the attack, the implementation of the attack was performed by Christian Catalano and Andrea Chezzi. Testing and validation were conducted by Christian Catalano, Andrea Chezzi e Vita Santa Barletta. Christian Catalano, Andrea Chezzi and Franco Tommasi contributed equally to writing the manuscript. The article was reviewed and revised by Franco Tommasi and Vita Santa Barletta, ensuring clarity and consistency. Christian Catalano supervised the research, providing guidance and oversight throughout the study.

Funding Open access funding provided by Università degli Studi di Bari Aldo Moro within the CRUI-CARE Agreement. This work was partially supported by the following project: SERICS. "Security and Rights In the CyberSpace—SERICS" (PE00000014) under the MUR National Recovery and Resilience Plan funded by the European Union—NextGenerationEU.

Data availability No datasets were generated or analysed during the current study.

Declarations

Conflict of interest The authors declare no competing interests.

Ethical approval In this research, we have designed cyber attacks on personal web account, Google account. These experiments have been performed against the author's accounts. All the experiments were performed by intercepting con- fiden- tial information belonging exclusively to the authors of the paper. This research does not involve human participants and/or animals.

References

1. Cyber security threat trends. (2021). https:// learn-cloudsecurity.cisco.com/umbrella-resources/umbrella/2021-cyber-security-threat-trends-phishing-crypto-top-the-list
2. Sharma, A.K., Galav, R.K., Sharma, B.: A comprehensive survey of various cyber attacks. In: 2023 6th International Conference on Information Systems and Com- puter Networks (ISCON), pp. 1–4 (2023). https://doi.org/10.1109/ISCON57294.2023.10111998
3. Patat, G., Sabt, M.: Please Remember Me: Security Analysis of U2F Remember Me Implementations in The Wild (2020). idHAL:hal-02865105
4. Ulqinaku, E., Assal, H., Abdou, A., Chiasson, S., Capkun, S.: Is Real-time Phish- ing Eliminated with FIDO? Social Engineering Downgrade Attacks against FIDO Protocols. Proceedings of the 30th USENIX Security Symposium (2021)
5. Schwartz, M., Machulak, M.: Securing the Perimeter: Deploying Identity and Access Management with Free Open Source Software. Apress, Berkeley, CA (2018). https://doi.org/10.1007/978-1-4842-2601-8
6. Alliance, F.: Certification overview. (2022). https://fidoalliance.org/ certification/
7. Microsoft: Microsoft digital defense report. (2022). https://query.prod.cms. rt.microsoft.com/cms/api/am/binary/RE5bUvv,https://query.prod.cms.rt.microsoft.com/cms/api/am/binary/RE5bUvv
8. M'Chirgui, Z.: The economics of the smart card industry: Toward-scoopetitive strategies. Econ. Innov. New Technol. **14**(6), 455–477 (2005)

9. Tommasi, F., Catalano, C., Taurino, I.: Browser-in-the-Middle (BitM) attack. Int. J. Inf. Secur. **21**(2), 179–189 (2021). https://doi.org/10.1007/s10207-021-00548-5
10. Catalano, C., Tommasi, F.: Persistent MobileApp-in-the-Middle (MAitM) attack. J. Comput. Virol. Hack. Tech. **20**(1), 27–39 (2023). https://doi.org/10.1007/s11416-023-00484-z
11. Kepkowski, M., Machulak, M., Wood, I., Kaafar, D.: Challenges with Passwordless FIDO2 in an enterprise setting: a usability study (2023). arXiv:2308.08096v2
12. Client to authenticator protocol (ctap). (2021). https://fidoalliance.org/specs/fido-v2.1-ps-20210615/fido-client-to-authenticator-protocol-v2.1-ps-20210615. html
13. W3C: Web authentication: An api for accessing public key credentials level 2. https://www.w3.org/TR/webauthn-2/
14. Web authentication: An api for accessing public key credentials level 2. (2022). https://www.w3.org/TR/webauthn-2/
15. Google: Chromium project. https://www.chromium.org/Home/
16. Yubico: Yubico product documentation. https://docs.yubico.com/
17. Feitian: Documentation resources. https://www.ftsafe.com/Support/Resources
18. Tzschoppe, J., L¨ohr, H.: Browser-in-the-Middle - Evaluation of a modern approach to phishing. EUROSEC '23: Proceedings of the 16th European Work- shop on System Security (2023). https://doi.org/10.1145/3578357.3589458
19. Jagannath, R.O., Jain, A.K.: Browser-in-the-middle attacks: a comprehensive analysis and countermeasures. Sec. Privacy **7**(5), 410 (2024)
20. Corinto, A.D.: Tre ricercatori italiani hanno scoperto come neutralizzare l'autenticazione a due fattori. (2022)
21. Browser-in-the-middle buca anche l'autenticazione a due fattori. (2022). https://www.securityopenlab.it/news/2016/ browser-in-the-middle-buca-anche-lautenticazione-a-due-fattori.html
22. Capec-701: Browser in the middle (bitm). (2023). https://capec.mitre.org/data/ definitions/701.html
23. MITRE: Mitre web reference. https://attack.mitre.org/
24. Schwartz, M., Machulak, M.: Securing the perimeter - Deploying Identity and Access Management with Free Open Source Software (Chapter 7: Strong Authen- tication). Apress Berkeley, CA, 231–265 (2018). https://doi.org/10.1007/978-1-4842–2601–8
25. Tommasi, F., Catalano, C., Corvaglia, U., Taurino, I.: MinerAlert: an hybrid approach for web mining detection. J. Comput. Virol. Hacking Tech. **18**(4), 333–346 (2022). https://doi.org/10.1007/s11416-022-00420-7
26. Tommasi, F., Catalano, C., Fornaro, M., Taurino, I.: Mobile session fixation attack in micropayment systems. IEEE Access **7**, 41576–41583 (2019). https://doi.org/10.1109/ACCESS.2019.2905219
27. Gonzalez-Burgue, A., Aparicio, D., Escobar, S., Meadows, C., Meseguer, J.: For- mal verification of the YubiKey and YubiHSM APIs in Maude-NPA (2018). arXiv:1806.07209v1
28. Jacomme, C., Kremer, S.: An extensive formal analysis of multi-factor authenti- cation protocols. IEEE 31st Computer Security Foundations Symposium (2018)
29. FIDO Security Reference - FIDO Alliance Implementation Draft 27 February 2018. FIDO Alliance (2018). https://fidoalliance.org/specs/fido-v2.0-id-20180227/fido-security-ref-v2.0-id-20180227.html
30. Sanam Ghorbani Lyastani, M.N.M.B. Michael Schilling, Bugiel, S.: Is fido2 the kingslayer of user authentication? A comparative usability study of fido2 assword- less authentication. IEEE Symposium on Security and Privacy (SP), pp 268–285 (2020)
31. Kentrell Owens, A.K. Olabode Anise, Ur, B.: User perceptions of the usability and security of smartphones as fido2 roaming authenticators. Seventeenth Symposium on Usable Privacy and Security (SOUPS 2021) - USENIX Association, pp 57–76 (2021)
32. Reynolds, J., Smith, T., Reese, K., Dickinson, L., Ruoti, S., Seamons, K.: A Tale of Two Studies: The Best and Worst of YubiKey Usability. IEEE Symposium on Security and Privacy (2018)
33. Chezzi, A., Catalano, C., Tommasi, F.: Bitm+ attack: An improved bitm/mitb- based phishing attack capable of passing through and defeating the fido/ctap2 protocol and the w3c webauthn api. Mitb-Based Phishing Attack Capable of Passing through and Defeating the Fido/Ctap2 Protocol and the W3c Webauthn Api (2024)
34. Documentation, M.-M.W.: Web authentication api. (2019). https://developer.mozilla.org/en-US/docs/Web/API/Web Authentication API
35. Wikipedia contributors: WebAuthn — Wikipedia, The Free Encyclopedia. [Online; accessed 2023] (2023). https://en.wikipedia.org/w/index.php?title=WebAuthn&oldid=1180406727
36. GitHub - FIDO Alliance WebAuthn demo (webauthn-demo. FIDO Alliance (fido- alliance) (2020). https://github.com/fido-alliance/webauthn-demo
37. WebAuthn WORKSHOP: Authenticating your web like a boss. FIDO Alliance (2018). https://slides.com/fidoalliance/jan-2018-fido-seminar-webauthn-tutorial
38. GitHub - FIDO Alliance WebAuthn demo (webauthn-demo). nerocrux (2020). https://github.com/nerocrux/webauthn-demo
39. Corporation, M.: [ms-rdpewa]: Remote desktop protocol: Webauthn vir- tual channel protocol. (2022). https://learn.microsoft.com/en-us/openspecs/windowsprotocols/ms-rdpewa/68f2df2e-7c40-4a93-9bb0-517e4283a991
40. Wikipedia contributors: Node.js — Wikipedia, The Free Encyclopedia. [Online; accessed 18-November-2023] (2023). https://en.wikipedia.org/w/index.php?title=Node.js&oldid=1185467042
41. Node.js: About node.js. https://nodejs.org/en/about
42. Node.js: About documentation - node.js. https://nodejs.org/en/docs
43. Puppeteer: Puppeteer documentation. https://devdocs.io/puppeteer/
44. Google: Welcome to chromium. https://blog.chromium.org/2008/09/ welcome-to-chromium 02.html
45. Express.js: Fast, unopinionated, minimalist web framework for node.js. https://expressjs.com/
46. VentureBeat: Case study: How why to build a consumer app with node.js. https://venturebeat.com/dev/building-consumer-apps-with-node/
47. noVNC: novnc - the open source vnc client. https://novnc.com/info.html
48. nginx: nginx. https://nginx.org/en/
49. docs, M.: The WebSocket API (WebSockets) (2023). https://developer.mozilla.org/en-US/docs/Web/API/WebSockets API
50. hardock14: dorowu-docker-ubuntu-vnc-desktop. Docker Hub (2017). https://hub.docker.com/r/hardock14/dorowu-docker-ubuntu-vnc-desktop
51. hardock14: dorowu-docker-ubuntu-vnc-desktop. Github (2017). https://github.com/HarGit14/dorowu-docker-ubuntu-vnc-desktop
52. Window: localStorage property. MDN web docs (2023). https://developer.mozilla.org/en-US/docs/Web/API/Window/localStorage
53. ArrayBuffer. MDN web docs (2023). https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global.Objects/ArrayBuffer
54. Catalano, C., Chezzi, A., Santa Barletta, V., Corallo, A.: Securing web technol- ogy and navigation against phishing through cnn. In: 2023 IEEE International Conference on Metrology for eXtended Reality, Artificial Intelligence and Neural Engineering (MetroXRAINE), pp. 944–948 (2023). IEEE