




SecPassInput: Towards Secure Memory and Password Handling in Web Applications

Pascal Wichmann¹ , August See², and Hannes Federrath¹

¹ Security in Distributed Systems, Universität Hamburg, Hamburg, Germany
{pascal.wichmann,hannes.federrath}@uni-hamburg.de

² Computer Networks, Universität Hamburg, Hamburg, Germany
richard.august.see@uni-hamburg.de

Abstract. JavaScript does not provide web applications the ability to overwrite or clear variables of primitive types, such as strings, when they are no longer required. Applications instead need to rely on the garbage collector to eventually clear sensitive data from memory. When accessing input fields natively provided by the browser via JavaScript, their values are accessed through primitive type variables and thus affected by this limitation.

In this paper, we analyze how the popular browsers Chrome, Chromium, Firefox, Opera, and Edge handle input values in memory. We find that sensitive values almost always remain in memory several minutes longer than necessary.

We propose the JavaScript library SECPASSINPUT that simulates a non-native input for passwords. The library does not rely on variables of a primitive type, thereby giving web applications the ability to clear and overwrite values in memory. We evaluate the security benefits of SECPASSINPUT by measuring how long values remain in memory after they are no longer needed, finding that the on-screen keyboard of SECPASSINPUT guarantees immediate removal from memory after triggering SECPASSINPUT's clear operation.

Keywords: Secure Memory Handling · Secure Password Handling · Password Input · Web Browsers · JavaScript · Web Security

1 Introduction

Browsers and JavaScript allow web applications to address many security-sensitive tasks. Such tasks may operate on sensitive values, such as passwords and cryptographic key material. While access to these values is required during a specific task, they can usually be removed from the device and memory immediately after the task's completion.

Attackers who manage to get virtual or physical access to a victim's device can exfiltrate data stored in memory and on the disk. Previous work has shown that sensitive data may be recovered from memory [11] even after the device is

powered off [7] or rebooted [2]. Consequently, reducing the time sensitive data is stored on the device and in memory can significantly reduce the threat surface. In this paper, we analyze how web browsers treat passwords processed by web applications. We provide a JavaScript library `SECPASSINPUT` that can be used to explicitly clear and overwrite passwords in memory.

The way browsers handle variables of primitive data types such as strings do not give web applications any means to explicitly clear or overwrite such variables. For example, when assigning a new string value to a variable that already contains a string value, the value is newly allocated in memory and the reference in the variable is updated, retaining the old string value in memory for possible future reference. The web application has to rely on the garbage collector to eventually free the variable, and the respective memory page to be overwritten after reassignment by the operating system at some later time. There are no guarantees how quickly this will happen, so strings may remain in memory longer than necessary. The focus of our research lies on confidentiality, i.e., protecting sensitive data from leaking.

JavaScript provides byte array types such as the `Uint8Array` that allow direct access to byte values, including in-place overwriting values without copying. By using such data types, the web application can take care of overwriting sensitive data itself, thereby minimizing the time that it is kept in memory. However, values that are entered by the user are usually provided via the browser's native input fields, from which they are accessed via a string variable in JavaScript. To address this issue, we provide `SECPASSINPUT`, a JavaScript library that simulates an input element and directly uses byte arrays. Thus, `SECPASSINPUT` never stores the value in a string variable while handling the user input. This library allows web developers to protect the passwords of their users by reducing the time the passwords are stored in memory to the required minimum. Especially web applications with high security requirements can benefit from this, for example to securely derive cryptographic keys from a password or to implement a web-based password manager.

To summarize, we make the following contributions:

- We analyze how Chromium, Chrome, Firefox, Opera, and Edge handle values of native input fields in memory on Windows and Linux.
- We propose `SECPASSINPUT`, a JavaScript library providing a secure password input that allows applications to clear entered values from memory.
- We evaluate the security benefits of our `SECPASSINPUT` and compare it to native password inputs.
- We discuss the feasibility of a widespread adoption of our `SECPASSINPUT` by web application developers.

The remainder of this paper is structured as follows: In Sect. 2, we discuss related work on memory handling of applications and attacks on memory. Section 3 describes the methodology that we use for our analysis of the browsers' memory handling, discussing the results of our browser analysis in Sect. 4. We describe the threat model in Sect. 5, followed by a presentation of `SECPASSINPUT` in Sect. 6 which we evaluate in Sect. 7. Subsequently, we discuss our results in Sect. 8 and provide an outlook and concluding remarks in Sect. 9.

2 Related Work

In this section, we discuss related work that considers memory handling of applications and attacks that target or utilize memory.

2.1 Memory Content Recovery

Especially in the context of forensics, past research has analyzed ways to recover sensitive contents from memory. Maartmann-Moe et al. [11] describe approaches to recover cryptographic keys from memory. Their success depends on the system state, where a freshly booting system does not allow them to recover any keys, while they are able to recover keys at least partially from all other system states.

Halderman et al. [7] analyze the persistence of DRAM. They find that data can be recovered from DRAM several seconds after removing its power. This allows attackers to steal cryptographic keys and other sensitive data from memory without requiring special equipment, for example by rebooting the device into an attacker-controlled system.

Lee et al. [10] investigate how Android applications handle passwords in memory. They perform an initial preliminary analysis on a sample of 11 applications and find that all of them are vulnerable to leaking passwords through an unnecessary long retention in memory. One of the causes is the usage of Java's string data type, which is immutable and thus cannot be overwritten. They address these problems by providing a patched version of Android's TextView input, allowing developers to securely handle passwords in their applications' memory. Their solution for Android application memory security is very similar to our SECPASSINPUT for web applications.

2.2 Secure Memory Handling in Native Applications

Previous research has considered the secure handling of sensitive data in memory. Chow et al. [2] analyze the threat of data exposure through not overwritten memory values, finding that even a reboot may be insufficient to clear memory contents if the device is not fully powered off during a soft reboot. They propose a "secure deallocation" approach which overwrites memory contents with zeros at or shortly after the deallocation of corresponding heap or stack elements. Gondi et al. [5] propose a tool that transforms applications written in C so that they explicitly overwrite sensitive data in memory when it is no longer needed.

Göktas et al. [4] consider attacks that evade information hiding. In information hiding, sensitive memory content is "hidden" at random locations in a very large address space. The attack creates many new program threads to fill the address space, making it easier to locate other processes' memory locations.

Other approaches try to protect sensitive memory contents of applications through encryption [3, 6, 8]. For example, Götzfried et al. [6] provide a solution transparent to the process that encrypts all the process' memory contents with a key that is stored in a CPU register. Other related work provides access control on the hardware level through modifications to the instruction set [14].

2.3 Browser Memory Analysis and Vulnerabilities

Jensen et al. [9] propose the tool MemInsight that allows to analyze the memory handling of web applications and detect memory leaks in JavaScript applications, while also providing specific support for browser-based peculiarities such as the document object model (DOM). Others consider the memory behavior of web applications and methods to detect and debug memory leaks [12, 13, 16].

Wang et al. [17] analyze information leaks from the browser, which among others includes writing memory into swap or hibernation files. They propose a framework that can protect from several information leaks in the transport of the web application and its data to the browser as well as in the browser itself. To address the swapping of sensitive data, their framework frequently accesses sensitive JavaScript objects from within the web application, assuming that the frequent access prevents the operating system from moving them to swap.

3 Methodology

In this section, we describe our methodology that we used to evaluate the memory behavior of browsers. We have performed our evaluation on Linux (Debian “bullseye”) and Windows (Windows 10) and considered the browsers Firefox, Chromium, Chrome, Edge (only on Windows), and Opera.

We used a virtual machine (VM) to run the operating systems and the browsers. This allows us to capture all memory contents of the VM, independently of the operating system and possible access restrictions it enforces on memory regions. In addition, these captures include possible occurrences of sensitive values that are caused by the operating system running in the VM. Also, using a VM enables us to temporarily suspend its entire execution to take a consistent memory snapshot.

We performed evaluation runs in several configurations. Each configuration includes the operating system, the browser, the type of input to analyze (browser’s native password input, browser’s native plaintext input, or our SEC-PASSINPUT), and the processing mode. We investigated two processing modes: (a) entering the password and clearing it via backspace, and (b) instead pressing a button that derives a cryptographic key from the password through the Web Cryptography API and clears the input value afterwards by setting the input’s value to an empty string. In addition, the configuration includes the VM’s background activity and the amount of memory assigned to the VM. To increase the reliability, we repeated each evaluation run multiple times. Some evaluations had nearly no variations so we repeated them only two times. Other evaluations with higher variations in the results of the individual repetitions were repeated six times. A new password is randomly generated for each evaluation run.

We used a script to control the VM from the host through the automation framework PyAutoGUI [15] which can be used to control keyboard and mouse inputs. The script first launches the configured browser and opens the desired test website. In addition, a second tab is opened and navigated to a news website. Next, the script sets the focus back to the first tab and enters the randomly generated password into the designated input. After a delay of 90 s, the password

is cleared according to the configuration, i.e., through backspace or through pressing the button triggering a key derivation. After another delay, navigation within the same site is performed by pressing a link that leads to another page within the same origin. This causes a fresh page load, i.e., we do *not* use a single-page application (SPA) which would handle such a navigation through JavaScript. Next, a navigation operation to another news website outside of that origin is triggered by entering its URL into the URL bar of the tab. Afterwards, we close the first tab and after another delay close the browser.

During the evaluation runs, our script suspends the VM execution every five seconds and takes a full snapshot of its memory. In addition, after every performed action, such as clearing the password, our script takes a snapshot from the suspended VM immediately. In the background, the full memory snapshots are processed to find all matches of the password value in the encodings UTF-8 and UTF-16. The matches and any surrounding 1 MB of data are retained for further analysis. In addition, we store a copy of all memory ranges where a match was found earlier within the same evaluation run. The remainder of the memory snapshots is discarded.

To verify that the browser automation performed the intended operations, our script takes a screenshot of the VM display before every memory snapshot operation. We manually analyzed four significant screenshots of every evaluation run (password completely entered, password fully cleared, tab closed, browser closed). Some evaluation runs have been invalid, e.g., due to unexpected browser dialogs, lost focus, or similar reasons. We discarded and repeated these invalid evaluation runs.

4 Memory Handling in Browsers

In this section, we analyze how browsers treat native input values in memory. We used our methodology from Sect. 3 to measure the retention and number of occurrences in memory for values before and after they are removed.

Figure 1 shows the results for a native password input. The upper graph presents the results when clearing the password via backspace, i.e., the value is never accessed via JavaScript. In the lower graph, the password is accessed via JavaScript and used to derive a key with the Web Cryptography API, clearing the input value after this operation through JavaScript. All values are averaged across multiple evaluation runs with identical settings.

For all browsers, entering the password immediately causes several occurrences in memory, ranging from 3 with Firefox to more than 8 with Edge. When the password is cleared through backspace, the number immediately drops. For Firefox, the value is less than one, i.e., some evaluation runs did not retain any further match. However, at least for some runs, all evaluated browsers retain copies of the password in memory. For several browsers, copies of the password remain in memory even after the browser has been closed.

When processing the password via JavaScript, after triggering the processing and clearing, the number of copies decreases at most by a negligible amount. Only after navigating within the same site, copies are removed from memory, while still retaining significantly more copies than an unprocessed password.

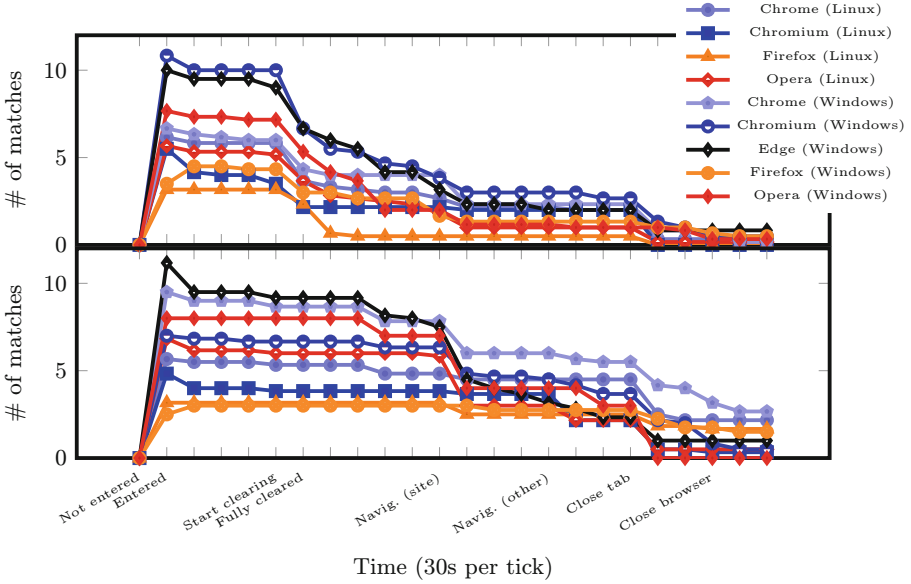


Fig. 1. Comparison of password retention in memory for native password input fields. Values are averaged across evaluation runs with identical configuration. The password is removed with backspace (upper graph) and used to derive a key using JavaScript and the Web Cryptography API (lower graph).

We also compared the browsers’ native password input (which displays dots and may indicate to the browser that its contents are particularly sensitive) and native plaintext input. Figure 2 shows the results for the plaintext input where the password is cleared via backspace and not processed with JavaScript. While the trends are similar to the password input above, all browsers produce significantly more copies in memory. This indicates that plaintext inputs provide a worse memory handling and are less suitable for sensitive inputs.

5 Threat Model

In this section, we discuss the threat model that underlies SECPASSINPUT. Within the scope of the browser, our threat model is based on the web attacker as defined by Akhawe et al. [1], i.e., an attacker that can visit web applications including those they target, host own web applications under different domains, and execute web applications in the browser of the victim. Beyond the browser, our attacker has limited local access to the victim’s devices, for example through an attacker-supplied native application that is run by the victim or through physical access. The attacker can read the memory of the browser process, which contains the variables of the interpreted JavaScript code. However, the attacker is only reading from memory at some points in time, rather than

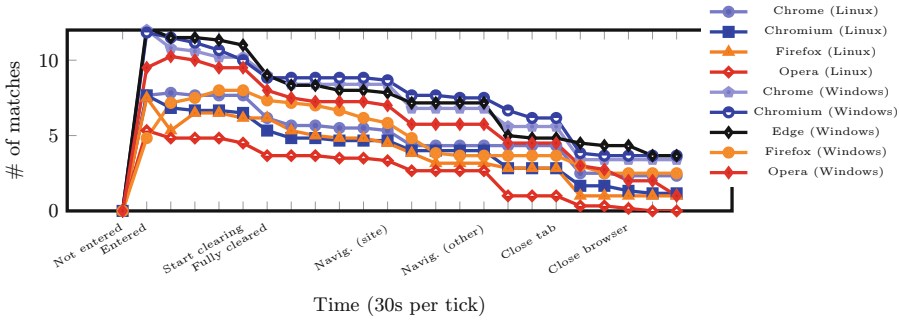


Fig. 2. Password is entered into *plaintext* input and removed via backspace.

constantly monitoring it for changes. In particular, we assume that the attacker does not read the browser’s memory after a user starts to enter a password and before it is cleared again.

While the attacker is able to *read* memory of other processes, they are not able to *modify* it. Neither can the attacker manipulate the program code or the behavior of the browser or the web application.

In the following, we describe four example scenarios to highlight the relevance of our threat model.

Cold Boot Attacks. A cold boot attack [7] can be used to extract memory contents of a running computer with physical access. If passwords remain in memory, a cold boot attack can recover such passwords when the attacker gains physical access to the still running computer later.

Malicious Application on Victim’s Device. The next scenario considers an attacker-controlled application running on the victim’s device with the goal of gaining information. Through that application, the attacker can read memory of the browser, either because the malicious application has administrative privileges or because its execution context allows to read memory of other processes of the same user. The application may for example take a memory snapshot at a regular time interval and send it to the attacker. We assume that the malicious application does not perform read operations while the user enters a password.

Swap and Hibernation Files. To extend the amount of available memory, many operating systems use an approach often called swapping, i.e., they move less-frequently accessed parts of memory contents to the disk. Unlike the physical memory, which loses its content usually a very short time after removing power, the disk is intended to persist data. Thus, when the swapped data is not encrypted and the device turned off, attackers may be able to extract sensitive information from it even a long time after the device was turned off.

The same applies to hibernation, or suspend-to-disk. There, all the contents of memory are stored to the disk, allowing to turn the device completely off



Fig. 3. Example of a SECPASSINPUT instance with activated on-screen keyboard

and restore the state of the device when it is powered on the next time. Unlike swapping approaches, suspend-to-disk stores a full memory dump that is not restricted to the less-frequently accessed parts of memory.

Crash Dumps. Operating systems may write crash dumps including parts of the browser’s memory contents into a file. The longer sensitive values are kept in memory, the more likely it gets that they are contained in such crash dumps.

6 Secure Password Input

In this section, we describe the functionality of our secure password input SECPASSINPUT. We provide a proof-of-concept implementation of SECPASSINPUT as a JavaScript library that is published on GitHub [18]. Figure 3 shows an example of a secure password input created with our library.

SECPASSINPUT relies on JavaScript’s `Uint8Array` data type, which allows explicit read and write operations for every byte contained in the array. Unlike JavaScript’s primitive string data type, which is always fully allocated in memory and immutable, the `Uint8Array` type contents can be cleared from memory by overwriting them in-place with a different value, such as zeros.

When accessing native browser input elements via JavaScript, their values are accessed through the `value` attribute, which provides a primitive `string` value that is allocated in memory and cannot be cleared by the web application. Also, besides JavaScript access, the browser stores additional copies of the inputs’ values in memory, which is outside of the control of the web application as well. Thus, we cannot rely on native input elements for SECPASSINPUT.

Instead, the SECPASSINPUT library is used to generate a secure input based on a regular `div` content element in the page’s DOM. The `div` element gets a tab index assigned, which allows user focus. Also, several child elements are dynamically added and manipulated while the user interacts with the secure input to display dots as character placeholders and a cursor that can be navigated using the arrow keys. SECPASSINPUT handles all keyboard events that are emitted while the `div` element is focused.

For each secure input, SECPASSINPUT maintains a `Uint8Array` that corresponds to the current value of the input. This array is initiated with a fixed length of 100 byte. An integer variable is used to store the actual length of the password. If the entered value exceeds the size of the array, a new array with the

size of the next multiple of 100 byte is allocated. The value is then copied into the new, larger array and the previous array securely overwritten with zeros.

A web application can access the password from `SECPASSINPUT` in two ways. Firstly, it can directly access a JavaScript object that contains the array with the password value and the integer variable containing its length. When using this method, the application itself needs to handle the length of the password, which may differ from the length of the byte array. Secondly, `SECPASSINPUT` provides a function that provides the application with a copy of the value in another array of the proper length. Using this method, it is the responsibility of the application to securely overwrite this copied array using another function provided by `SECPASSINPUT` as soon as this value copy is no longer required.

As an alternative to the preallocated array of a fixed size that is extended when needed, it would be possible to implement `SECPASSINPUT` with an array of a fixed size that is replaced by a new array on every input operation by the user. In that case, every change of the value would require a new array to be allocated, the old array value to be copied, and the old array to be securely overwritten. At the same time, that implementation would not require new copies when the application needs a properly-sized array for processing. However, we assume that the most common use case for `SECPASSINPUT` is a user entering a password once, causing a lot of change operations of the password value, followed by the processing by the web application, which requires a very low number of accesses to the password. In particular, the password is usually not updated during multiple processing steps, which means that a single copy of the value can be used in all steps. For this use case, our implemented approach of the fixed-sized array is the better trade-off.

Each keyboard event that corresponds to a printable character or a delete operation causes an update of the internal state of the secure password input as explained before. Besides printable characters, `SECPASSINPUT` maintains a cursor position to support navigation within the input. This allows users to navigate within the input field using the arrow keys, optionally combined with the `Ctrl/Command` key to move to the start or end of the input.

On-Screen Keyboard. `SECPASSINPUT` provides an on-screen keyboard to use mouse or touch inputs rather than key presses. The on-screen keyboard is displayed underneath every secure input field when activated by the user.

Developers that use `SECPASSINPUT` can enforce using the on-screen keyboard for individual secure input instances, i.e., disable handling of input from the regular keyboard. This can provide additional protection, for example from keyloggers that capture all physical keyboard input on a victims' device.

Disabling Visual Feedback. By default, `SECPASSINPUT` displays dots that correspond to the number of characters entered, including a cursor indicating the current position at which typed characters are inserted. This behavior is identical to browsers' native password input fields, which minimizes the risk of user confusions. However, to increase the security of the secure inputs in the presence

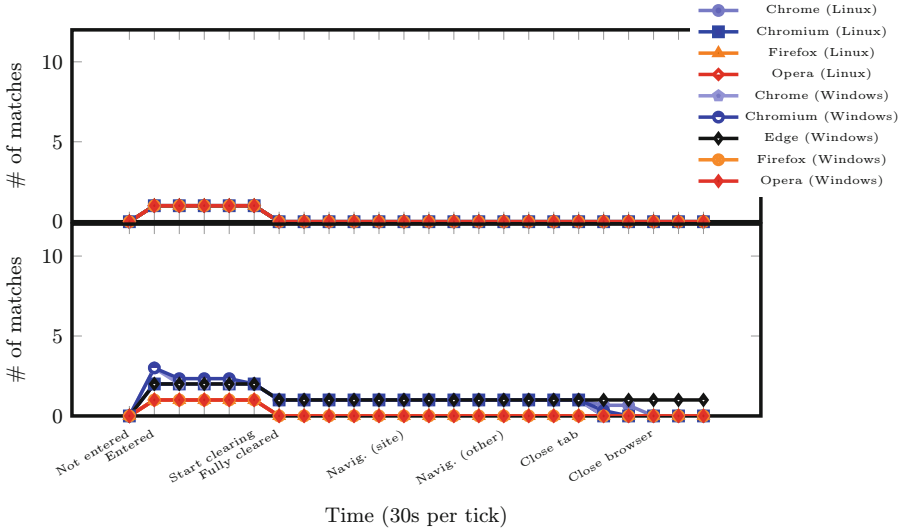


Fig. 4. Memory occurrences of passwords entered through SECPASSINPUT: via on-screen keyboard (upper graph) and virtual USB keyboard (lower graph).

of attackers that observe the user’s screen while typing a password, SECPASSINPUT offers an option that disables the displaying of dots. When activated, the input does not give any visual feedback while the user is typing, preventing the password length to be exposed through the visual state of the input.

7 Evaluation of SECPASSINPUT

To evaluate the effectiveness of our secure password input, we tested our proof-of-concept web application in Google Chrome, Chromium, Firefox, Opera, and Edge and verified that the password is immediately overwritten in memory.

Figure 4 shows the results for SECPASSINPUT when entering a password using the on-screen keyboard (upper graph) and the regular keyboard (lower graph). With the on-screen keyboard, all browsers in both operating systems cause only a single occurrence of the password in memory which is immediately gone after clearing the input. Thus, when using the on-screen keyboard, SECPASSINPUT guarantees that the password is immediately removed completely from memory as soon as it is no longer needed.

When using a regular keyboard, Chrome, Chromium, and Edge cause 2 and up to 3 memory occurrences of the password, some of them remaining after clearing the password. Only Opera and Firefox exhibit the desired behavior of causing a single memory occurrence that is immediately gone after the input is cleared. Thus, using a regular keyboard that causes key presses handled by the operating system and browser leaks the password in some of the tested browsers, implying that full protection is only possible when using the on-screen keyboard.

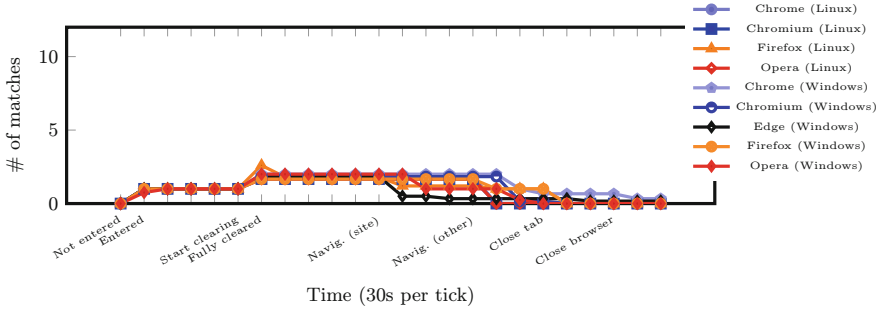


Fig. 5. Memory occurrences of password entered via SECPASSINPUT with on-screen keyboard. A key is derived via the Web Cryptography API.

In Fig. 5, we present the results for SECPASSINPUT using the on-screen keyboard and deriving a key using the Web Cryptography API. Until clearing the password, there is no difference to the evaluation without processing. After the clearing operation, which is preceded by the key generation operation, at least one and up to two additional copies are caused by the key derivation operation. These remain in memory much longer than necessary, being cleared only half a minute after navigation within the same site (Edge) or even after navigation to an external site. For Chrome on Windows, one match remained even after closing the browser in some of the evaluation runs. Thus, using the Web Cryptography API causes additional copies of the handled value which are outside of the control of SECPASSINPUT.

8 Discussion of SECPASSINPUT

In this section, we discuss SECPASSINPUT and our evaluation methodology. Also, we discuss aspects that need to be considered before deploying the library.

Password Processing. The security benefit of SECPASSINPUT depends on the purpose of the password input and the further processing by the web application. If the password is sent to the server in plain text, it needs to be written to a native JavaScript string variable, voiding most of the benefits SECPASSINPUT aims to provide. As our evaluation revealed, the derivation of a cryptographic key via the Web Cryptography API can also leak the password into long-retained copies in memory in some browsers.

One solution can be provided through incorporating cryptographic operations into SECPASSINPUT. For example, the library can be extended with functionalities to derive cryptographic keys from the password without exposing it to any APIs or functions that may cause additional, not securely cleared copies of the password.

Potential for Wrong Usage by Users. SECPASSINPUT may cause confusion for new users, as it does not support functionalities of native inputs, such as clipboard pasting or password manager filling. Consequently, users may accidentally input the password into unintended fields. This can be addressed by visible, clear warnings below SECPASSINPUT inputs.

Accessibility Software. As SECPASSINPUT relies on a simulated non-native input, it does not support text input features, including most password managers and accessibility software. Thus, in the current architecture, it may be inaccessible to some people. As a workaround, SECPASSINPUT can be extended with an insecure fallback to a native input. While this voids the security advantages of SECPASSINPUT for the affected users, it solves its accessibility issues.

Because browsers do not recognize SECPASSINPUT as an input field, touch-based devices do not display the keyboard. SECPASSINPUT's on-screen keyboard is touch-compatible, resolving this issue.

Native Browser Support for Clearing Values. SECPASSINPUT is a JavaScript library and thus limited to the capabilities the language provides within the browser. Thus, it is not always guaranteed that this approach is sufficient to remove all copies of the password from memory in every environment, especially when the value is used for further processing. As the evaluation showed, while SECPASSINPUT works reliable in some browser environments, it does not guarantee the passwords to be fully removed from memory in all browsers and scenarios.

A more reliable approach would be native browser functionalities that allow applications to securely erase values from memory and handle passwords in a secure way, or to mark values as sensitive so they are securely overwritten by the browser automatically. However, SECPASSINPUT can be deployed without further requirements on the client side, thus fills the gap and builds a foundation for further work to provide such functionality natively in the browser.

Regarding values processed via JavaScript, browsers can add a method to allow web applications to mark values such as passwords as sensitive. Such values can then be cleared by the browser in a memory-secure way.

On the operating-system level, a security mechanism can be introduced that allows processes to be marked as sensitive. The operating system can then take care of securely overwriting the processes' memory as early as possible, including immediate overwrites of deallocated memory values.

Relevance of the Threat Model. The attacker considered in our threat model has very strong capabilities, including access to most parts of memory while the password is not entered. This can be used to read any data handled by the web application, for example cryptographic keys derived from the password or session identifiers in cookies or the local storage. Consequently, protecting the password from an attacker may not be sufficient if sensitive data derived from it still leaks to the attacker.

To address this limitation, the web application can further reduce the time such sensitive data is contained in memory, for example by repeating the key derivation every time the key is required. However, this increases the number of times that the password is required, which may provide a larger threat surface for attacks on the password input.

To protect from exploitation of leaked session identifiers, IP-based access restrictions can be enforced. Thus, extracting such session identifiers from the victim's device is less critical than extracting the password, which allows to arbitrarily request new valid session identifiers.

9 Conclusion

In this paper, we analyzed the retention time of sensitive data in memory of popular web browsers. Our research revealed that passwords entered in browsers may stay in the computers' memory for a long time even after closing the browser and almost always remain in memory longer than required.

As a protective measure, we proposed SECPASSINPUT, a JavaScript library that allows web applications to explicitly clear password input values from memory. We implemented a proof of concept of SECPASSINPUT as a JavaScript library. Our evaluation showed that SECPASSINPUT with its on-screen keyboard can guarantee that passwords are immediately cleared from memory as soon as they are removed from the input in all tested browsers. Using APIs such as the Web Cryptography API in some browsers causes additional copies in memory that cannot be cleared by the web application nor SECPASSINPUT.

As future work, we intend to research more extensive protective strategies, which may be incorporated directly into the browser or operating system. Also, regarding usable security, we intend to integrate password advice and password strength validation into SECPASSINPUT.

References

1. Akhawe, D., et al.: Towards a formal foundation of web security. In: 23rd IEEE CSF 2010, pp. 290–304 (2010)
2. Chow, J., et al.: Shredding your garbage: reducing data lifetime through secure deallocation. In: 14th USENIX Security 2005 (2005)
3. Enck, W., et al.: Defending against attacks on main memory persistence. In: 24th ACSAC 2008, pp. 65–74 (2008)
4. Göktas, E., et al.: Undermining information hiding (and what to do about it). In: 25th USENIX Security 2016, pp. 105–119 (2016)
5. Gondi, K., et al.: SWIPE: eager erasure of sensitive data in large scale systems software. In: 2nd CODASPY 2012, pp. 295–306 (2012)
6. Götzfried, J., et al.: RamCrypt: kernel-based address space encryption for user-mode processes. In: 11th Asia CCS 2016, pp. 919–924 (2016)

7. Halderman, J.A., et al.: Lest we remember: cold boot attacks on encryption keys. In: 17th USENIX Security 2008, pp. 45–60 (2008)
8. Henson, M., Taylor, S.: Memory encryption: a survey of existing techniques. *ACM Comput. Surv.* **4**, 53:1–53:26 (2013)
9. Jensen, S.H., et al.: MemInsight: platform-independent memory debugging for JavaScript. In: 10th ESEC/FSE 2015, pp. 345–356 (2015)
10. Lee, J., Chen, A., Wallach, D.S.: Total recall: persistence of passwords in Android. In: 26th NDSS 2019 (2019)
11. Maartmann-Moe, C., Thorkildsen, S.E., årnes, A.: The persistence of memory: forensic identification and extraction of cryptographic keys. *Digit. Investig.* **6**, S132–S140 (2009)
12. Pienaar, J., Hundt, R.: JSWhiz: static analysis for JavaScript memory leaks. In: CGO 2013, pp. 11:1–11:11 (2013)
13. Rudafshani, M., Ward, P.A.S.: LeakSpot: detection and diagnosis of memory leaks in JavaScript applications. *Softw. Pract. Exp.* **1**, 97–123 (2017)
14. Shi, W., et al.: InfoShield: a security architecture for protecting information usage in memory. In: 12th HPCA-12 2006, pp. 222–231 (2006)
15. Sweigart, A., et al.: PyAutoGUI (2021). pypi.org/project/PyAutoGUI/
16. Vilk, J., Berger, E.D.: BLeak: automatically debugging memory leaks in web applications. In: 39th SIGPLAN 2018, pp. 15–29 (2018)
17. Wang, F., Mickens, J., Zeldovich, N.: Veil: private browsing semantics without browser-side assistance. In: 25th NDSS 2018 (2018)
18. Wichmann, P.: SecPassInput: secure password input library (2023). <https://github.com/wichmannpas/sec-pass-input>