Empirical Studies on the Security and Usability Impact of Immutability

Sam Weber
New York University
6 Metrotech Center
Brooklyn, NY
Email: samweber@nyu.edu

Michael Coblenz, Brad Myers, Jonathan Aldrich, Joshua Sunshine
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA, US
Email: {mcoblenz, bam, aldrich, sunshine}@cs.cmu.edu

***Abstract*—Although it is well-known that API design has a large and long-term impact on security, the literature contains few substantial guidelines for practitioners on how to design APIs that improve security. Even fewer of those guidelines have been evaluated empirically.**

Security professionals have proposed that software engineers choose immutable APIs and architectures to enhance security. Unfortunately, prior empirical research argued that immutablity decreases API usability.

This paper brings together the results from a number of previous papers that together aim to show that immutability, when carefully designed using usability as a first-class requirement, can have positive effects on both usability and security. We also make observations on study design in this field.

I. Introduction

It is well-known that API and language design can have a large impact on security, and this barrier is difficult, if not impossible, to overcome by training alone. For example, buffer overflows were understood and documented as early as 1972, but are still one of the most common vulnerabilities [1]. As another example, Wang et. al. [2] showed that usability problems in Facebook's and Microsoft's online authentication interfaces resulted in security vulnerabilities in most of the websites that used them. Furthermore, APIs are typically designed by a small number of experienced developers but have an extremely long life-span, and therefore the impact of poor API design can have far reaching consequences.

Usability issues in security-relevant APIs, such as cryptography or authentication libraries, have a clear impact on security. In this work we focus instead on APIs that are not specifically security-related because programmers using them are not actively considering security and therefore are more likely to be susceptible to the conceptual failings and unstated assumptions that underlie most vulnerabilities.

Unfortunately, most existing work on API design concentrates on relatively low-level features, such as documentation and method naming. We would like to encourage research on higher-level API design features. Herein, we'll discuss in detail the current state of knowledge regarding one such feature: immutability.

A. Immutability

Immutability, the prevention of state changes, is widely recommended in both programming language and security documents such as Oracle's *Secure Coding Guidelines for Java SE* [3]. Immutability automatically prevents TOCTTOU (Time of Check To Time of Use) attacks and, in general, is said to aid programmer understanding of potential system behavior by eliminating the need to reason about object modifications [4].

For example, in an early release of Java, the method `Class.getSigners()` returned an array containing the signers of a class. Unfortunately, what was returned was a reference to the actual internal array containing the signers, not a copy of it. Since arrays are mutable, malicious code could alter the system's knowledge of code signers [5]. Thus the concern about mutability affecting security is more broad than just TOCTTOU attacks.

Unfortunately, previous empirical research, described in detail in Section II, indicated that APIs that create immutable objects are less usable by programmers.

In this paper we summarize two previously published papers ([6] and [7]) which together further investigated the usability and security impact of immutability and limitations on state-changes in general. In particular, these papers aimed to examine the discrepancy between the community opinion of immutability and experimental findings, determine if security and usability were, in fact, opposed to each other and, above all, provide evidence-based advice to practitioners about whether immutability was a desirable API feature.

This body of work includes:

- An interview study with expert software engineers to find out how practitioners control state in large, complex software projects, their perceived needs and how they feel about existing language features and tools. It was determined that they strongly desired improvements in their ability to control state. Design recommendations were extracted.
- A review of existing literature and programming languages resulted in a classification system for mutability restrictions in programming languages.
- A language extension to Java, called Glacier, was designed and implemented. Glacier allows users to express transitive class immutability, which guarantees that immutable objects cannot reference any mutable state [7].

978-1-5386-3467-7/17 $31.00 © 2017 IEEE
DOI 10.1109/SecDev.2017.21

Create-Set-Call Style:

```
var foo = new FooClass();
foo.Bar = barValue;
foo.Use();
```

Required Constructor Style:

```
var foo = new FooClass(barValue);
foo.Use();
```

Fig. 1. Alternate Object Creation Styles [8]

- Evaluations of Glacier, done both by retrofitting existing applications and by laboratory experiments, demonstrated that it could be applied to real applications and that programmers were more effective at specifying immutability and modifying code in immutable classes using Glacier than using Java's `final` keyword.

Additionally, this paper also provides observations about experimental methodology in this field.

II. Related Work on Immutability and API Usability

Stylos and Clarke, in *Usability Implications of Requiring Parameters in Objects' Constructors* [8], investigated the usability of two alternate styles of object creation, illustrated in Figure 1. If an object, in order to be valid, needs a "Bar" value, the first style, "Create-Set-Call" has the programmer instantiate an empty object and then explicitly sets a "Bar" field with the appropriate value. The second style, "Required Constructor", has the programmer supply the "Bar" value in the constructor.

Although not explicitly stated in this paper, the relationship to immutability is obvious: an immutable object cannot be created using the Create-Set-Call style.

Stylos and Clarke recruited thirty different professional programmers, none of whom were students. Each participant executed six tasks. In the first task they designed an API for accessing files. The following tasks involved using classes of each style to: access files, use provided classes, debug code, initialize the inventory of an online-store application, and read code on paper. Finally, semi-structured interviews were conducted to ascertain in more detail the reasoning behind the participants' actions and their opinions on API design.

To their surprise, programmers strongly preferred and were more effective in using create-set-call style APIs. Commonly create-set-call was seen as more flexible. Even programmers who highly valued correctness did not feel that required constructors offered any assurances about the validity of an object.

These results motivated the authors of [6] to do further work investigating whether there was a tradeoff between security and usability of immutable APIs, summarized in the next section.

Of course, there has been much other research on immutability and API Usability. Immutability has long been a feature of functional languages: purely functional languages support no directly mutable data structures, and other functional languages encourage designs that minimize the use of mutable state. In addition, functional approaches can eliminate large classes of security bugs in applications, as in Ur/Web [9], which prevents all injection attacks (and other classes of security flaws). Other recent languages such as Rust, Clojure, Scala and Swift have incorporated functional concepts. Designers of the PLT family of languages suggest that immutability is helpful for web development: a stateless programming style avoids various session-related bugs [10]. However, we know of no empirical data on the effect of functional approaches on security. Significant differences in many language design decisions, as well as differences in application spaces and user populations, have so far made it difficult to separate the immutable aspects of functional languages from other aspects when considering potential security impact.

In the area of API Usability, a recent survey [11] shows an increasing amount of research and commercial interest in this area, and some studies have explicitly linked lower usability of APIs to less secure applications. For example, a study by Fahl et al. [12] of 13,500 popular free Android apps found 8.0% had misused the APIs for the Secure Sockets Layer (SSL) or its successor, the Transport Layer Security (TLS), and were thus vulnerable to man-in-the-middle and other attacks; a follow-on study of Apple iOS apps found 9.7% to be vulnerable. Causes include significant difficulties using security APIs correctly, and Fahl et al. [12] recommended numerous changes that would increase the usability and security of the APIs.

Similarly, Acar et al [13] investigated the usability of crytographic APIs, determining empirically how the design of such an API affects the security of the systems that use them. One difference between this work and the work described here is that crytographic APIs directly involve security, whereas immutability indirectly affects security.

III. Interview Study

In order to determine how practicing software engineers think about system state and what issues they face, Coblenz et. al. obtained IRB approval and conducted semi-structured interviews with a sample (N = 8) of software engineers at several US- and Europe-based organizations. They focused on software engineers who worked on large software projects, and had at least seven years of professional experience, with a mean of fifteen years. Typically participants had worked on projects with millions of lines of code and hundreds of people. Space limitations prevent a complete description of this study, but more details can be found in [6].

Participants almost universally reported that incorrect state changes were a significant source of bugs. One participant ranted:

> ... my favorite is where you have data that is supposed to be immutable and is only settable once in theory but that's not well enforced and so it ends up

> getting re-set later either because it gets re-initialized or because someone is doing something clever and re-using objects or you have aliasing where two objects reference the same other object by pointer and you make changes...

Another participant reported that immutability was a key part of his system's architecture: "By design, we've made the key data structures immutable. We never update or delete any data objects. This ensures that an error or problem can never put the system into an undesirable state."

Immutable classes were reported to be very frequently used, but that the languages that participants used provided very poor support for them. For example, C++'s `const` was described as being inflexible and did not provide the desired functionality. Programmers complained that declaring a method `const` requires transitively annotating all methods that the first method calls, and that this was a serious burden.

IV. Mutability Restrictions

One problem with attempting to address programmers' desires for immutability and the control of state changes in general is that there are many different concepts and dimensions to the control of state that are not clearly differentiated.

In [6], a classification for mutability restrictions in programming languages is presented, which has eight distinct dimensions. For example, *assignability* restrictions disallow assignment. In Java, if a class `A` has a field `f` declared `final`, then this is an assignability restriction: in all objects of class `A`, `f` can never refer to a different object than the one with which it was initialized. In contrast, the C declaration `const int *x` provides what is called a *read-only* restriction: `x` is a pointer to an `int` and might later refer to a different address, but the reference `x` cannot be used to change the value at any memory location to which `x` points.

The dimensions *scope* and *transitivity* are particularly noteworthy. Scope refers to whether a restriction refers to particular objects or to all of a class's instances. Transitivity refers to whether restrictions apply only to the direct object being considered or to all objects that are referred to by that object. As described above, Java's `final` as applied to fields is non-transitive: a particular field in a particular instance can only refer to a single object, but the state of that object might change. A transitive immutability restriction would not only prevent that field's reference from changing, but the state of that object and all objects referred to by it directly or indirectly would likewise be prevented.

The participants interviewed in the previously-descripted study seemed to desire transitive immutability restrictions that applied to all instances of a class.

V. Glacier

As described in detail in [7], one of the authors (Coblenz) implemented a Java extension, called Glacier, that provided transitive class immutability features to Java. Glacier supports the type annotation `@Immutable` to indicate that instances of that type are transitively immutable.

Glacier is a static typechecker that takes advantage of Java's annotation support so that it can be parsed by standard Java parsers. Standard Java development tools can be passed arguments that will cause the Glacier processor to be invoked appropriately.

In Glacier, a class that is declared to be immutable must have all of its fields immutable and fields cannot be assigned to outside the class's contructors. If a reference to an object is not immutable, then there are no immutability guarantees: at runtime the object may or may not be immutable.

Design decisions to take into account such issues as subclasses, primitives and arrays had to be made — see the referenced paper for details.

VI. Usability of Glacier

In order to evaluate the usability of Glacier, Coblenz et al. conducted several usability studies (see [7] for full details).

ZK Spreadsheet is a commercial Java spreadsheet which supports importing Excel documents. The authors of this application stated that they did not use any immutable data structures because they were concerned about the cost of copying that may result from immutability. However, cell styles can be shared between cells, and there is no tracking of which cells use the same style. As a result, when ZK Spreadsheet modifies a cell's style it has to make a new style object instead of modifying the existing style. Coblenz refactored the spreadsheet's code to use Glacier in approximately twenty hours. In the process two previously unknown bugs were found in the application. One of these bugs was in a copy method, which was no longer necessary in the immutable version; the other was in a code that improperly used a cache. These were reported back to the authors of *ZK Spreadsheet*, who acknowledged the issues and fixed one of them (finding that the other occurred in code that was never invoked).

Additionally, Coblenz et al. recruited twenty experienced Java programmers for a laboratory user study in which participants either used Glacier or Java's `final` functionality to implement immutability. Users of Glacier were given training on it while users of `final` were given training on how to implement immutability using it. Four tasks were given to participants, and each task was time-limited. One of those tasks was explicitly designed to mimic the conditions that resulted in the `getSigners()` bug that was described in Section I-A. The study results demonstrated that fewer implementation errors occurred when using Glacier compared to when using standard Java features. In particular, on the task designed to replicate the `getSigners()` bug, all of the Glacier users who finished performed the task correctly while half of the `final` users who completed the task recreated the security bug. This provided evidence that Glacier is more effective at preventing security problems than `final`.

VII. Discussion

Based upon our interview and Glacier usability studies, we maintain that APIs that support immutability can be both more usable and secure, given reasonable language support.

However, we do have to consider why the Stylos and Clarke study [8] came to the opposite conclusion.

One possibility is when the studies were done. Stylos and Clarke conducted their work in 2005, whereas our work was done between 2014 and 2016. One hypothesis is that the rise of web development led the developer community to consider ways to control the problems arising from the web's massive concurrency. The same time period also saw a more widespread use of functional concepts in programming languages. As a result, the concept of immutability may have become much more prominent over time, and the decade between our studies and theirs might be significant.

Another consideration is that the Stylos and Clarke did not explicitly call-out immutability and primarily considered utility classes – classes that were not part of an application's core. The one exception to this, their shopping cart task, was a small codebase and done over a very short time period. In contrast, our professional interviews explicitly asked our participants to consider their major projects and their core data structures. It is entirely reasonable for programmers to have different priorities for their core data structures than their utility classes. Finally, Stylos and Clarkes study focused on learnability and initial usability, rather than long-term concerns such as robustness and security.

This points out a general issue with lab usability studies. Although the results are nicely quantitative, the costs involved require that participants can only spend a relatively short time doing tasks and that they are unlikely to be invested in the quality of the code that they produce. This might not be an issue when considering utility code or tools, but we urge caution in applying the results to critical code or tasks. Interview studies can be used to verify the applicability of results obtained from laboratory studies.

VIII. Conclusion

In this paper we have summarized the research results investigating the security and usability impacts of immutability in APIs. In general, although the Software Engineering community has started to investigate the impact of API design decisions on usability, we feel that the security community likewise needs to play a role. This work on immutability is a prime example, as the recent work described herein was motivated by the observation of the security impacts of the earlier Stylos and Clarke paper.

IEEE's Center for Secure Design has made a number of design recommendations for creating more secure code [14]. We maintain that empirical investigation of many of these recommendations, such as the avoidance of reflection, would be beneficial to practitioners. For example, reflection exists because it is useful, and finding techniques that satisfy programmers' use-cases without leading to security vulnerabilities would be valuable.

We also wish to champion the use of qualitative studies in this field. Laboratory studies are often prized because they provide quantitative results. However, they do have the problem that the tasks involved have to be limited in both time and number of participants. Studies have shown that security bugs have different characteristics than other kinds of bugs (see [15] and [16]), and these differences seem to manifest themselves more in larger, more complex systems. Qualitative studies can be used to ensure that issues that arise in these systems are not overlooked.

Acknowledgment

We are grateful for the assistance of Forrest Shull and Whitney Nelson on the prior work. This research is supported in part by NSA lablet contract #H98230-14-C-0140 and by NSF grant CNS-1423054. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect those of any of the sponsors.

References

[1] Y. Younan, "25 years of vulnerabilities: 1988-2012," *RSA Conference*, 2013.

[2] R. Wang, Y. Zhou, S. Chen, S. Qadeer, D. Evans, and Y. Gurevich, "Explicating SDKs: Uncovering Assumptions Underlying Secure Authentication and Authorization," in *USENIX Security*, 2013, pp. 399–414.

[3] Oracle Corp. (2015) Secure coding guidelines for the java se, version 4.0. [Online]. Available: http://www.oracle.com/technetwork/java/seccodeguide-139067.html

[4] H. Abelson and G. Sussman, *Structure and Interpretation of Computer Programs*. MIT Press, 1996.

[5] G. McGraw and E. W. Felten, *Securing Java*. Wiley, 1999.

[6] M. Coblenz, J. Sunshine, J. Aldrich, B. Myers, S. Weber, and F. Shull, "Exploring language support for immutability," *Proceedings of the 38th International Conference on Software Engineering - ICSE '16*, pp. 736–747, 2016. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2884781.2884798

[7] M. Coblenz, W. Nelson, J. Aldrich, B. Myers, and J. Sunshine, "Glacier : Transitive Class Immutability for Java," *39th International Conference on Software Engineering (ICSE'17)*, pp. 496–506, 2017.

[8] J. Stylos and S. Clarke, "Usability Implications of Requiring Parameters in Objects' Constructors," *29th International Conference on Software Engineering (ICSE'07)*, pp. 529–539, may 2007. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4222614

[9] A. Chlipala, "Ur/Web: A simple model for programming the web," in *ACM SIGPLAN Notices*, vol. 50, no. 1. ACM, 2015, pp. 153–165.

[10] S. Krishnamurthi, P. W. Hopkins, J. McCarthy, P. T. Graunke, G. Pettyjohn, and M. Felleisen, "Implementation and use of the plt scheme web server," *Higher-Order and Symbolic Computation*, vol. 20, no. 4, pp. 431–460, 2007.

[11] B. A. Myers and J. Stylos, "Improving API usability," *Communications of the ACM*, vol. 59, no. 6, pp. 62–69, 2016.

[12] S. Fahl, M. Harbach, H. Perl, M. Koetter, and M. Smith, "Rethinking SSL development in an appified world," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*. ACM, 2013, pp. 49–60.

[13] Y. Acar, M. Backes, S. Fahl, S. Garfinkel, D. Kim, M. L. Mazurek, and C. Stransky, "Comparing the Usability of Cryptographic APIs," *IEEE Symposium on Security And Privacy*, 2017.

[14] I. Arce, K. Clark-Fisher, N. Daswani, J. DelGrosso, D. Dhillon, C. Kern, T. Kohno, C. Landwehr, G. McGraw, B. Schoenfield, M. Selzer, D. Spinellis, I. Tarandach, and J. West. (2014) Avoiding the top 10 software security design flaws. [Online]. Available: https://www.computer.org/cms/CYBSI/docs/Top-10-Flaws.pdf

[15] F. Camilo, A. Meneely, and M. Nagappan, "Do Bugs Foreshadow Vulnerabilities? A Study of the Chromium Project," *The 12th Working Conference on Mining Software Repositories*, 2015. [Online]. Available: http://dl.acm.org/citation.cfm?id=2820551

[16] S. Zaman, B. Adams, and A. Hassan, "Security versus performance bugs: A case study on firefox," in *Proceedings of the 8th Working Conference on Mining Software Repositories*, 2011.