A Brief Aetiology Of Computer System Security

Callan Christophersen
Dept. Computer Science
The University of Auckland
Auckland, New Zealand
cchr158@aucklanduni.ac.nz

***Abstract*—Security is one of the biggest issues facing the IT industry today, 2017, and has been for many years. It is also likely to be at the forefront of our concerns for many years to come. In this paper we examine if the perspective of B.W. Lampson in "Computer Security in the Real World" [16] still holds 13 years later by looking at this topic through the use of two studies of vulnerabilities in two complex systems with large user bases. They are "Finding and Preventing Bugs in JavaScript Bindings" by Brown et al which examines bind code bugs in Chrome's V8 engine [5], and "An In-Depth Study of More Than Ten Years of Java Exploitation" by Holzinger et al which is a taxonomical study of Java exploits [14], as well as three well documented exploit classes XSS, SQLi, and CSRF. Fundamentally the problem is who should bear the burden of responsibility when systems fail, the user or the provider? Why is this problem seemingly intractable as it is intrinsically the same type of problem that has been faced by older engineering disciplines.**

I. Introduction

The Global Risks Report 2016 [1] cites that the second greatest concern of CEOs in 2015 was cybersecurity. Given the scale and costs of this problem it might be expected that decade old vulnerabilities, for which solutions are well known [24], [26]–[29], would be all but non-existent so attackers would not have an easy foot hold. Unfortunately this assumption would be wrong. In his article "Computer Security in the Real World" B.W. Lampson states "Most computers are insecure because security is expensive" [16], combine this with a general lack of security awareness among managers and developers [1] and it isn't difficult to see why.

Although there are vulnerabilities that can exist solely in hardware [13], most exploited vulnerabilities occur in software [1], [18], [19], [21], [28], [32], [39], [40], [45]. Broadly, there are two types of code: functional code, and code that accounts for erroneous user input [38]. Functional code is what is taught in most entry, and even mid-level, university Computer Science and Software Engineering Courses. It is said that this code fulfils the functional requirements of the client. It in no way accounts for if the user types a character instead of an integer, supplies a number that is too large, or does any one of a multitude of different things that could cause the application to crash. Code that accounts for erroneous user input, checks for such actions by the user and redirects the control flow of the application to correct for this. This type of coding is not usually taught until functional coding has already been mastered. When developers are learning how to write functional code it is common practice for them to use online resources such as tutorials to help develop their applications. These tutorials, like the 64,000 PHP tutorials found by Unruh et al [41], work from a functional perspective but contain very insecure code. For most amateur developers, once the functional code has been written, error checking code is added to or around it. It is not very common for the functional code to be changed very much, if at all, during this transition. Therefore, any insecurities in the functional code are likely to be present in the finished product as well. In addition, insecurities can also be introduced in the error checking code compounding this problem.

The combination of Lampson's observation and the prevalence of bad coding practices has led to much of today's software being filled with security bugs [18], [19], [21], [28], [32], [39], [40], [45]. A quick read of Microsoft, Apple, IBM, etc end user agreements will tell you that the software is provided "as is" and is used at the user's own risk. It could be argued that those software companies which continuously produce suspect software will lose customers in the long run due to their damaged reputations. However for three consecutive years (2013-2015) iOS was responsible for over 80 per cent of mobile vulnerabilities [39], [40] but people still lined up for the latest iPhones. Brown et al in "Finding and Preventing Bugs in JavaScript Bindings" [5] found 81 vulnerabilities in the Chrome browser but Chrome is still used by millions of people. In their paper "An In-Depth Study of More Than Ten Years of Java Exploitation" Holzinger et al [14] found nine generic weaknesses that break the Java security model leading to 61 unique exploits but Java is still executing on billions of devices. Therefore if market forces will not motivate companies to fix these issues, a shift in culture and education is needed. Bad habits formed early on, have led to many university graduate developers being unable to write secure code [9]. This, combined with the fact that writing secure code is a preventative measure that does not provide any direct benefit other than blocking potential losses later on, means that the managers of these developers have not made secure code a priority either [38].

These aforementioned affects have led to statistics like: cybercrime cost US$445 billion in 2014, which has subsequently increased [19]; more than 430 million new

unique pieces of malware in 2015, up 36 percent from 2014; and approximately 429 million identities were exposed in 2015 of which more than 90 million of those came from just 9 breaches [39]. Here we argue that although Lampson's observation from 2004 still holds today, we can do better, so why don't we? To help answer this we look at the two works documenting vulnerabilities in Chrome [5] and Java [14]. We compare these complex vulnerable systems with three other old classes of vulnerabilities with well known solutions to try to uncover why vulnerabilities with known solutions still persist.

II. Why it's hard to secure systems

Lampson also provides us with a framework for what "computer security" looks like. He breaks security systems into 3 aspects: specification/policy, implementation/mechanism, and correctness/assurance. Put simply what is the systems purpose, how is this realised, and does it work? In terms of policy Lampson tells us that secrecy, integrity, availability, and accountability should be our main concerns. This is slightly different from the traditional confidentiality, integrity, and availability [35]. The first three of Lampson's policy headings are isomorphic with Russell and Gangemi's aspects of computer security but Lampson's accountability requirement is not. By accountability Lampson means "knowing who has had access to information or resources". This last policy doesn't seem to generalise as well as the first three.

Lampson suggests five broad implementations of these policies. They are: isolate, exclude, restrict, recover, punish. Lampson puts these in order of decreasing levels of security. In this case to isolate is to keep everyone out, to exclude is to discriminate against the "bad guys", to restrict is to limit the damage that can be done, to recover is to replace/repair that which was damaged, to punish is to hurt the attacker making the exploit more expensive.

When it comes to assurance, Lampson places the heaviest burden of responsibility on the developers of these systems "Developers also need a process that takes security seriously, values designs that make assurance easier, gets those designs reviewed by security professionals, and refuses to ship code with serious security flaws". Lampson's reasoning here is that both users and administrators are very resilient to education and therefore if the goal is to create secure systems then, security must be design driven.

Here we present two examples of non-trivial systems that have had serious vulnerabilities exposed in them. These examples are illustrative of both the problems faced when securing real world systems and some of Lampson's points about the difficulties of "real world security". We will also see how Lampson's security framework fits with these systems.

A. Finding and Preventing Bugs in JavaScript Bindings

Binding code is a layer of code that handles that translation of value representation, error propagation, and types between a "high level" language, usually a dynamically typed language, and its runtime system, which is normally written in a "low level", statically typed language. In the case of Chrome the high level, dynamically typed, language is JavaScript and its runtime system is V8 which is written in C++, a statically typed language.

Along with enhanced system features, there are security advantages to this design. Many low level security bugs such as buffer overflows, use-after-frees, and memory leaks can be abstracted away from the developer and managed by the runtime in a secure way. However this assumes that the runtime is capable of dealing with such bugs. The main reason that the authors give for the occurrence of these security flaws is "Binding code has the dangerous distinction of being both hard to avoid and hard to get right". Brown et al appear to be in agreement with Lampson here in that both agree that software is complicated, and in this case, necessary.

The problem of binding code is particularly pronounced in JavaScript because of the extreme flexibility of the language. The binding code in the V8 runtime even allows call backs from the binding layer to the JavaScript code. For instance the toPrimitive property allows a developer to define the behaviour exhibited by a variable when it is accessed. When such a variable is passed to the binding layer and the runtime attempts to retrieve its value, a call back to the JavaScript is required. This behaviour is important because it gives an attacker a greater attack surface.

Brown et al [5] describe how the binding code of V8, and runtime systems like it, can be vulnerable to crash-, type-, and memory-safety vulnerabilities in their binding code.

- Crash-safety: These bugs occur when input, given to the V8 runtime, causes the C++ code to hard crash or segfault. The result of this type of crash is a loss of control over the information flow. The most obvious use for this vulnerability is a denial of service attack. However this type of bug can also be used to break language level security abstractions leading to covert channel attacks and information leakage. For instance, in the event of a segfault, the OS will take a core dump which could contain sensitive information. If the attacker can still execute code on the machine, possibly via a redirect prior to the segfault, then this core dump could be read by the attacker.

  This type of bug was, in fact, introduced with the runtime systems like V8, in their binding code since JavaScript itself is crash-safe.

- Type-safety: Because type mismatches can be used to crash the system, all of the problems with crash-safety bugs are present with type-safety bugs as well. Type-safety bugs occur when a pointer/reference of type t points to an object in memory of type u and t is not compatible with u. This can allow an attacker to carry out a type confusion attack which could allow remote code execution. The result of this could be system wide penetration.

  Since this type of bug only occurs because of the difference in type systems between JavaScript and C++, this bug is a result of the V8 runtime.

- Memory-safety: As type-safety violations are a form of memory violation all of the problems with type-safety are present with memory-safety bugs as well. Memory-safety violations occur when an attacker can avoid the safety checks in binding code and read/write to memory that is restricted. For instance if an attacker is able to cause a segfault in the course of the core dump and the attacker is able to control the program counter then, the attacker could execute arbitrary code that is in memory.

  As with the other two bugs this one has nothing to do with JavaScript itself. Since JavaScript cannot access memory without preserving abstraction, it can only access the memory that has been allocated by its runtime.

Although Brown et al do not provide evidence for their claim "it would be remarkable if these languages did not contain essentially identical flaws. Therefore, we believe that other high-level language runtimes [...] stand to benefit from lightweight checkers and more principled API design", it is likely that other JavaScript runtime systems, such as Mozilla's Rhino engine, will share many of these vulnerabilities with V8.

Brown et al also argue that vulnerabilities in binding code are worse than vulnerabilities introduced by a JavaScript developer at the application level. This makes sense since an adversary wouldn't have to create an application specific attack they could create one attack that would affect all applications using JavaScript.

Brown et al developed static code checkers along with an API to find bugs like these in V8's code, so the developers can patch these problems, and build V8 code in a secure way. As with all new security technology this API isn't battle hardened nor did Brown et al formally test it.

1) Fitting V8 into Lampson's Framework: At its most basic level Chrome is a web browser, created by Google, that runs on personal computers. As such, what is to be kept confidential is the users private communications, the integrity we want to protect is that of the data stored on the users machine, and the availability we wish to preserve is the access to web applications. However it isn't immediately clear who is to be held accountable for Lampson's fourth criterion. Is the user accountable for a breach in the security model? Are the creators of the web application? Is it Google, who created the web browser? Or is it the attacker that managed to inject some malicious JavaScript into the web application? The way Lampson talks about accountability is in the context of auditing an institution like a business, but this isn't very applicable here since browsers don't keep logs to that level of granularity and the malicious code is executed in a way that appears it was intended all along.

What about Lampson's 5 implementations? Isolation doesn't apply since the browser will connect to any web application the user wants. Exclusion is applicable since the user will try not to connect to bad web applications. Restriction is applicable since this is what the browser tries to do through sandboxing its processes. Recovery is applicable if the user maintains backups. Punishment is not applicable as neither browser or users have the authority to do this.

Overall, most of Lampson's framework is applicable to this system. However the pieces of Lampson's framework that aren't as applicable are the pieces where he deviates from the standard security model. This is possible because browsers are quite different from other types of software in that they are programs that essentially download arbitrary code from an unknown source(s) and run it locally. Nevertheless, it could be said that this is a bit of an edge case for Lampson's framework.

B. An In-Depth Study of More Than Ten Years of Java Exploitation

Holzinger et al document a decades worth of security vulnerabilities against Java [14]. They break these vulnerabilities into three categories: single-step–, restricted-class–, and information hiding– attacks. These three categories contain 9 weaknesses or attack vectors.

To make things clear, the authors break each vulnerability they investigate into final goal, attack vector, primitive, attack primitive, helper primitive, implementation, and exploit; in order of abstraction. The *final goal* is the end objective of an attack and is why it is considered malicious. An *attack vector* is a set of intermediate steps or goals needed to realise the *final goal*. These intermediate steps or goals are labelled as *primitives* by the authors. *Attack primitives* are a subtype of *primitive* defined as a *primitive* that breaks the security model of the target system without necessarily achieving the *final goal* state by itself. *Helper primitives* do not break the security model of the target system, are a counterpart to one or more *attack primitives*, and if absent, would render the *attack primitive* redundant. *Implementations* instantiate

primitives. An *exploit* is an instance of an *attack vector*.

The 9 weaknesses described by Holzinger et al are:

- Caller sensitivity: This refers to certain methods that should only be accessed by trusted code. The authors describe this weakness as using an intermediate trusted function to avoid the permissions check and access the sensitive code.
- Confused deputies: This type of weakness is used to invoke caller sensitive methods. The intermediate trusted function in the caller sensitivity weakness is a confused deputy. This type of weakness can be generalised as any trusted guard that can be confused, allowing untrusted code to access trusted code. In this case allowing untrusted code to route a call sequence through a system class.
- Privileged code execution: The authors hold this type of attack as distinct from confused deputy attacks since, here an attacker can execute code that passes the privilege checks and does not depend on caller sensitivity. That is, the attackers code is executed with a higher level of privilege than it should be.
- Loading of arbitrary classes: This vulnerability allows an attacker to load classes that their code should not be able to access such as the system class.
- Unauthorized use of restricted classes: This can be either defining a custom class in a privileged context or accessing field values or access modifiers when they are private.
- Unauthorized definition of privileged classes: This is a means of achieving arbitrary code execution by way of defining a class in a trusted domain.
- Serialization and type confusion: This weakness can, for instance, be used to instantiate a classloader defined ahead of time by the attacker. This can then be used to create classes with higher privileges.
- Reflective access to methods and fields: If reflection has been used improperly in system classes or caller sensitive methods then this can be used to bypass information hiding.
- MethodHandles: This is similar to the previous weakness except it uses methodHandles instead of reflection.

These weaknesses are not entirely discrete. For instance many of the exploits involving the restricted classes as either the final goal or as part of the attack vector also involve a confused deputy and caller sensitive method. This suggests that most of the exploits studied by Holzinger et al could be solved by applying a few conceptually, and in some cases technically, simple patches. For instance removing caller sensitivity from public interfaces for classloading. However this would break backwards compatibility making this a very undesirable fix for both Java developers and users alike.

In light of this, Holzinger et al also seem to agree with Lampson when they state "... the Java sandbox is actually anything but a simple box. Instead it comprises one of the world's most complex security models in which more than a dozen dedicated security mechanisms come together to–hopefully–achieve the desired isolation." [14]. It would also be reasonable to point out that the exploits themselves are not trivial in their own right.

1) Fitting Java into Lampson's Framework: Just as with the previous example, the first three of Lampson's specifications apply here. The JVM aims to keep parts of itself (restricted classes) and parts of its host system confidential or secret. The integrity of the host system, restricted access classes and variables, as well as the JVM itself must be preserved. The availability of the host system and the JVM must also be maintained. In this case it is also clearer which party is accountable for the security. Oracle is responsible for developing and assuring the Java security model and therefore are the accountable party.

As for Lampson's mechanisms, the JVM is not isolated as Lampson meant it, because it has to allow untrusted code to execute and it has to interact with the host system. The Java security model does try to exclude untrusted code from its trusted code. The JVM does operate as a sandbox and therefore restricts untrusted code according to Lampson's definition. One of the ideas behind Java's sandbox is that it doesn't matter if things go wrong in the sandbox so long as it can be thrown away and restarted. In this sense the JVM is recoverable, however if the developer has not backed up the state of the JVM then, this cannot be recovered. The JVM has no capacity to punish attackers.

Lampson's framework seems more appropriate to this example then it does to the previous example as it fits more of Lampson's specifications and is more in line with his mechanisms.

III. Lampson lives on

In "Computer Security in the Real World" [16] Lampson talks about many other important aspects of system security. Such as: local access control, where an access control list is used to authenticate users, normally with a password; distributed access control, which has the same principle as local access control but occurs between multiple autonomous systems, with special care taken to encrypt any

communication; trust chains, where one trusted node passes trust to one other node and then another and so on forming a chain. Although the papers discussed above do use some of these aspects, for simplicity and because they are not what is being attacked directly, we do not discuss these ideas further. These ideas are well established and researched by the system security community and the specific technologies, such as Kerberos [2], have been updated. For these reasons we concentrate on Lampson's ideas of how to view security systems.

Lampson sums up the two aspects of the above example systems which have led to many of these bugs: "First, software is complicated, and in practice it's impossible to make it perfect. Even worse, security must be set up, and [...] setup is complicated too. Second, security gets in the way of other things you want. [...] security interferes with features and with time to market." [16]. These examples are large development efforts, with many stakeholders, resulting in software which can be deployed in almost any environment. From this perspective Lampson appears to have found a reasonable upper bound for what we can expect from "real world" system security.

Both examples fit, to varying degrees, Lampson's framework and this framework does offer some ways forward. For instance neither system has a satisfactory punishment mechanism. If, the law permitting, such a mechanism were to be developed then, an amnesty may be reached with attackers, given that some kind of mutually assured destruction principle would now be in play.

IV. Why we can do better

Lampson was right when he said "Practical security balances the cost of protection and the risk of loss" and "The bad guys balance the value of what they gain against the risk of punishment". In other words both sides perform a cost benefit analysis. This, in addition to his points about the complexity of systems and the setup cost of security systems contributing nothing to the useful output, makes the task of securing any non-trivial system seem truly daunting. Even Google is not immune to this effect. When Lampson's point of view is taken in light of the fact that the companies responsible for developing these systems do not bear the brunt of the cost for security flaws, one can see that these companies are not incentivised very well to build secure systems. As Milton Friedman put it "nobody spends somebody elses money as carefully as he spends his own" [11]. If risk is the product of the asset's cost with the probability of an exploit occurring then, regulating the 'tech' industry in a similar way to other engineering industries, where companies are punished for not meeting minimum safety standards, would increase the cost and therefore increase the risk of an exploit to the company developing the system. The cost to the companies for a reasonable standard of security wouldn't be too great either since for many bugs the fixes are free, maintain backwards compatibility, and are well documented for ease of implementation. In other words, without regulation it is cheaper to do the bare minimum in terms of security. However it would not require heavy handed regulation to tilt the economics in favour of better security.

V. Some bugs are fixable

Some, particularly those in the employ of these development companies, may think that the above is excessive. However there are many security flaws that have been around for a very long time, have well known solutions and yet, still persist in causing havoc to this day. Below are three such examples from the OWASP top 10 list [28]. Part of the reason for this is developers have not done as Lampson suggests and "[refused] to ship code with serious security flaws". After all there is little incentive for them to do this and a large employment incentive for them not to. Next we examine three security flaws that have existed for almost as long as the first commercial web applications. We look at their root causes and accepted defences.

A. Cross Site Scripting (XSS)

XSS attacks are a type of code injection which have existed since at least 1999 [33] and are a particularly dangerous and wide spread type of attack. For this reason the Symantec Internet Security Threat Report 2016 [39] places this attack at number 5 on its list of top ten unpatched vulnerabilities, and at the time of writing, XSS is at number 3 on OWASP's top 10 most critical web application security risks [28]. These attacks are so dangerous because of how easy it is to detect XSS vulnerabilities in applications and how difficult it can be to secure an application against them. Once the code has been injected into the web application, the user's web browser has no way of knowing that this code did not come from the otherwise trustworthy application. Thus the web browser will execute the injected code with the same privilege level as any other code sent to it by the same entity.

XSS attacks can occur when data enters a web application through an untrusted source. This is most commonly a web request but can be through other means such as a page with mixed content i.e. a page that has content loaded from many different sources; some using a HTTPS [8] connection, and others not. These attacks can also occur when data is included in dynamic content that is sent to a user without being validated for malicious content. For example manipulation of the Document Object Model (DOM) objects on the client side or insecure scripts being executed by the application server [26]. From these two attack vectors three different types of XSS vulnerability can be derived: persistent XSS, reflected XSS, and DOM based or client side XSS.

a) Persistent XSS: sometimes referred to as stored XSS [26], occurs when a vulnerable application stores user input without validating it first. This type of XSS is less common than reflected XSS but is considerably more damaging since the injected code affects all users visiting that section of the site and will continue to do so until it is manually removed. Applications and websites which allow content to be shared between users are most vulnerable to this type of XSS. Some examples of these sites are:

- Social networking sites
- Blogs
- Message boards and forums
- Collaboration services like github.
- Webmail clients
- Enterprise resource planning or customer relationship management applications

b) Reflected or non-persistent XSS: is the most prevalent and pervasive category of XSS attack. These attacks are so called because they are "reflected" off the web server. In other words, a request with some malicious code in it is sent to the web server and the response from the web server contains that injected malicious code which is then executed by the client's web browser as code from a trusted source [26].

In order for a malicious third party to use a reflected XSS attack effectively malicious code is duplicitously injected into a single HTTP request [23]. The attacker can then URL-encode that HTTP request and devise some scheme to have the client send it to the web server. For this reason reflected XSS attacks are usually combined with another style of attack such as a phishing attack.

c) DOM based XSS: attacks are the least common of the three categories of XSS attacks [15]. These attacks can be impossible for an application or server to detect since no data is required to leave the victim's browser allowing them to elude most detection techniques. DOM XSS attacks are made possible by the insecure use of DOM objects by the client's browser i.e. the use of DOM objects that is not fully controlled by the server side logic.

It is important to note that placing full control of DOM objects on the server side is easier said than done. This security requirement may come into conflict with certain design decisions made by developers for ease of coding and satisfying the client's requirements. For instance, one might have a client-server application with some dynamic content on certain pages where the content of the page changes based on the session information. It is considerably easier, in a development team, for a front end developer to write some JavaScript on the client side to manipulate the page contents, believing that this session information is safe if the connection is under HTTPS [8], than it is for the front end developer to have to communicate this design feature to a back end developer, requiring the two developers to coordinate their implementations.

In a DOM based XSS the attacker takes advantage of the fact that it is the user's web browser that will populate the DOM objects, like document.location, based on the browser's point of view and not the server's. The user's web browser can populate DOM objects with user defined URL parameters and if those objects are then used in an execution context of the application, any code injected into those URL parameters may be executed there. This is the crux of a DOM based XSS and is why full control over the DOM must be maintained by the web application. A DOM based XSS vulnerability is present if the site uses data from any of the DOM objects in an insecure way.

In a HTML document there are good places and bad places to put untrusted data[1]. These input fields should be encapsulated in a data context so that whatever the input is, it is never executed as code. Most of this encapsulation is achieved through using the correct character escaping techniques for each part of the HTML document. In general a Whitelist approach is preferred because it is secure by default. That is all things allowed by the list are known to be safe and all things not allowed by the list are prevented from progressing as soon as they are detected whether they are safe or not. Thus all non-safe actions are prevented. OWASP has produced a Whitelist of rules to mitigate XSS attacks [31]. This list follows industry standards and the preponderance of the literature [20], [37], [42], [43] on prevention of XSS attacks. Following these guidelines will significantly reduce the attack surface with respect to XSS.

B. Structured Query Language injection (SQLi)

SQLi attacks have been known since before the year 2000 [36] and are similar to XSS attacks in that they are both command injection attacks. SQLi attacks occur when a SQL query is executed directly by way of user input into the application [29]. These queries will then return private information from the database to the client, modify the database's contents, or run commands on the database such as creating a new database or revealing other databases. The most common case is for a SQL injection to modify a legitimate SQL query that the application would have otherwise made. SQLi attacks come in three distinct flavours: inband SQLi, out-of-band SQLi, and inferential SQLi attacks. Their descriptions are below.

a) Inband SQLi: are attacks that occur entirely within a single channel of communication. In other words, the attacker enters a SQL query into an input field, that query is executed by the database and the results are returned to the attacker through their client.

[1]Here untrusted data means data that could contain a malicious payload.

b) Out-of-band SQLi: attacks also fit into another category of attack called cross channel scripting (XCS) attacks [4]. Out-of-band SQLi attacks are launched in one communication channel and whatever is returned by the database after the query has been executed arrives to the attacker through a different channel. Out-of-band SQLi attacks are especially easy to combine with other attacks [10] such as XSS exploits, insufficient authentication vulnerabilities [22], (distributed) denial of service attacks [17], and domain name server hijacking [44].

XCS attacks such as out-of-band SQLi attacks are described by Bojinov et al as "Detecting an XCS or reverse XCS vulnerability can be difficult because these attacks abuse the interaction between the web interface and an alternate communication channel. Simply inspecting the web application code and the other service code is not enough to detect the vulnerability. The web application and the other service, such as an FTP server, can be completely secure in isolation and become vulnerable only when used in conjunction." [4]. Despite the complexity of this bug solutions are possible.

c) Inferential SQLi: according to Chandrashekhar et al "[Inferential SQLi is] where the attacker does not actually receive any direct data in response to injection attempts. Instead, the attacker has to infer the data present by reconstructing data based on variations in responses from the application" [7]. This is not the same as inband SQLi because the database does not actually return anything necessarily; nor is it akin to out of band SQLi since all the communication can occur along a single channel. This is why it is the most sophisticated of the three types of SQL injection [7].

Given that two out of the three different types of SQL injection attack are at least very difficult to detect in general, it is very important that developers implement logically sound defences to SQL injections. A determined attacker or group of attackers could create site specific tests to discover and exploit vulnerable sites. Thankfully there are such logically sound defences and they are quite easy to implement. These are [30]:

- Prepared Statements (Parameterized Queries), by creating predefined SQL queries, with variables to be bound to user input later, the developer ensures that an attacker cannot change the intention of the queries. So even if an attacker enters a malicious SQL payload, this payload will be treated as text just like a legitimate query.
- Stored Procedures, pre-defined queries that are stored in the database ahead of time.
- Whitelisting and Escaping user input, these should be used as a last resort but only known safe values should be allowed and all user input should be escaped using the BDMS escaping scheme.

If developers apply these strategies and best practices in a coherent design, the resulting application will avoid the vast majority of SQL injection flaws.

C. Cross Site Request Forgery (CSRF)

There are some misconceptions about defences for CSRF attacks [24]. These misconceptions arise from a fundamental misunderstanding about how CSRF attacks are carried out. Many purported defences such as "secret" cookies (which are included in every request by the user and can be copied by the attacker) and multi-step transactions (which the attacker can emulate) do *not* in fact work [3].

Cross site request forgery is the exploitation of certain browser behaviours to hijack an unsuspecting user's session with a server [12], [24]. This class of attack was named in 2001 but was known about prior to 2000 [6]. The main objective is to make the user's web browser send state changing requests to the web server or impersonate the user by reusing their session information. In contrast to the other types of attacks discussed in this section, CSRF does not exploit the trust the user has in the web server, but rather the trust the web server has in the user. It is worth noting here that if XSS is present then CSRF is implied because any XSS vulnerability can be used to gather and manipulate the session information of a user in spite of any other CSRF defence.

A successful CSRF attack requires certain information about the session in progress in order to be successful:

1) Browser behaviour regarding the handling of session information.
2) Knowledge of valid web application URLs. Potentially within the authentication space[2].
3) Information known by the browser relied upon for session management. For instance basic authentication as opposed to form based authentication.
4) (Optionally) Existence of HTML tags which require a HTTP resource. Such as the image tag. This is an especially "nice to have" element for an adversary[3].

It should be noted that in the case of item 3 form based authentication is considered more secure because instead of a

[2]Authentication space here has the meaning: The set of all URLs that a user must be authenticated to access.

[3]This is nice to have because if content is not loaded under HTTPS then an attacker can replace the HTTP response with their own. Meaning that an attacker can inject JavaScript directly into the site without needing prior contact with the client.

digest of the user's credentials being passed back and forth as in basic authentication, the server uses the credentials entered in the form fields to create a unique session key which is invalidated when the user logs out. With basic authentication, if the web browser does not properly close the session by clearing the user's credentials when they logout then the user's session can be revived by simply resending the digest to the server in a HTTP request.

Items 1, 2, and 3 are individually necessary and together sufficient for a successful CSRF attack. It should be noted that all CSRF attacks are blind. That is the attacker cannot see what the response from the target site looks like. Therefore CSRF attacks almost always use state changing requests since it does not benefit the attacker to simply request data on behalf of the user. As an example, if an application relies upon information that would be known to the browser; while simultaneously engaging in behaviours such as basic authentication then, a CSRF is possible since an attacker can, in most cases, discover application URLs with relative ease.

The general best practices for defending against CSRF are to implement all XSS defences, check standard HTTP headers to verify the request's origin, and check CSRF tokens, provided they have been managed correctly [25]. The Origin header was invented to help prevent CSRF [3]. This can be achieved by comparing the Origin header's field with the target origin. The Origin header would be sufficient on its own to verify the origin except there are situations where the Origin header is null, as in the case of cross origin redirects. In this case the Referer header must be used to verify that the hostname matches the target origin. Ideally both checks should be performed and if either fails then the request should be discarded. In the cases where the Origin header will be null then the Referer header check must be performed.

a) Lampson asks a similar question: "Will things get better?" he answers this in the affirmative by stating that the market may change its priorities but only after a calamity has occurred. The basic idea here being the level of risk that users *profess* they are prepared to accept before a security breach is much higher than the level of risk they say they will accept after such a breach has taken place. Does progress in the security industry have to look like this? Could users and administrators not simply be reasoned with to implement better security before a catastrophe?

D. A Reasonable Level Of Security

It would be incorrect for the reader to infer that the above details "perfect" defences against the mentioned classes of vulnerabilities. However these vulnerabilities remain on watchdog top vulnerability lists because such basic steps are not taken to secure web applications [28], [39]. As was shown in Section II not all systems are simple to secure, nor would it be cheap to do so. However, what the above tells us is some security issues do, in fact, have well known, simple, solutions that can be implemented in a cost effective manner. As a concrete example lets take another look at the first example in Section II about binding bugs. All those binding bug exploits started with JavaScript. Hardening a web application against XSS as described above would not only help to stop XSS exploits but also help defend against malicious JavaScript code that targets binding code bugs from being injected. Therefore a *reasonable* level of security is somewhere between Lampson's upper bound for real world security and the relative ease with which systems like ASP.NET's Linq library deals with XSS and SQLi threats on behalf of the developer. Given that a vulnerability in one system may represent a threat, not just to itself or its users but to other systems as well, it seems reasonable that a regulation mandating certain minimum security standards for IT systems is warranted in the same way that adventure tourism; engineering; food processing; manufacturing; and farming all seem to need regulations.

However, unlike adventure tourism and other forms of engineering, system security, along with the rest of its industry, grows and changes at a much faster rate. By the time a body has written regulations, gone through a reviews process, had committee meetings, heard public submissions, heard public appeals, etc, the technology the regulations were suppose to regulate will have evolved. For this reason any regulation must focus on returning the onus for building reasonable secure systems back to the systems' creators and not focus directly on the implementation of the systems themselves. At the very least it would discourage software companies from shipping code like that found in the 64,000 PHP tutorials mentioned earlier [41] and would help to fulfil one of Lampson's goals for developers.

There are of course many more difficult questions around regulating the development of secure systems, such as sovereignty, fairness, the extent and power to be held by the regulating agency(ies), the security requirements for a given system in a given context, etc. Notwithstanding these challenges the prodigious resilience users have to education may prove to be insurmountable in its own right. Having said that if our university graduates cannot write secure code [9] then, maybe education is part of the problem. If light weight regulations were introduced and the security awareness of developers is increased from an early stage in their training to a reasonable level as above then, simpler more secure systems should follow.

VI. Conclusion

As we have shown security is still as big an issue today as it was when Lampson wrote his article "Computer Security in the Real World" in 2004. Cybercrime cost US$445 billion in 2014, and has subsequently increased [19]; more than 430 million new unique pieces of malware were discovered

in 2015, up 36 percent from 2014; and approximately 429 million identities were exposed in 2015 of which more than 90 million of those came from just 9 breaches [39]. In their paper Brown et al [5] found 81 vulnerabilities in Chrome's binding code and Holzinger et al [14] found nine generic weaknesses that break the Java security model neither of which is an indication that the aforementioned statistics will improve. In addition to this basic XSS, SQLi, and CSRF vulnerabilities are still among the most ubiquitous vulnerabilities in web applications [6], [28], [36], [39]. It is hard to deny that the catastrophe Lampson said was required for things to improve has happened without claiming that such a catastrophe must be singular and/or existential in nature.

In spite of this, we have not seen the change in the IT industry that is needed to address these security issues, not because users haven't shifted their priorities but because users do not have more than a superficial understanding of what they are buying. As Donald Rumsfeld put it "As we know, there are known knowns. There are things we know we know. We also know there are known unknowns. That is to say we know there are some things we do not know. But there are also unknown unknowns, the ones we don't know we don't know" [34]. So why don't we do better? As long as the cost burden of exploits remains on users and users remain largely ignorant of the problems, the developers of these systems will not have the incentive to change. If the IT industry wants to be taken seriously like other engineering fields then, it needs to mature and take responsibility for itself.

REFERENCES

[1] The global risks report 2016. *"World Economic Forum"*, 2016.

[2] Sufyan T Faraj Al-Janabi and Mayada Abdul-salam Rasheed. Public-key cryptography enabled kerberos authentication. In *Developments in E-systems Engineering (DeSE), 2011*, pages 209–214. IEEE, 2011.

[3] Adam Barth, Collin Jackson, and John C Mitchell. Robust defenses for cross-site request forgery. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 75–88. ACM, 2008.

[4] Hristo Bojinov, Elie Bursztein, and Dan Boneh. Xcs: Cross channel scripting and its impact on web applications. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, pages 420–431, New York, NY, USA, 2009. ACM.

[5] F. Brown, S. Narayan, R. S. Wahby, D. Engler, R. Jhala, and D. Stefan. Finding and preventing bugs in javascript bindings. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 559–578, May 2017.

[6] Jesse Burns. Cross site request forgery. *An introduction to a common web application weakness, Information Security Partners*, 2005.

[7] Roshni Chandrashekhar, Manoj Mardithaya, Santhi Thilagam, and Dipankar Saha. Sql injection attack mechanisms and prevention techniques. In *International Conference on Advanced Computing, Networking and Security*, pages 524–533. Springer, 2011.

[8] T. Dierks and C. Allen. The tls protocol version 1.0. RFC 2246, The Internet Engineering Task Force, January 1999.

[9] EC-Council Foundation. The talent crisis in infosec. Technical report, EC-Council Foundation, 2013.

[10] Nadya ElBachir El Moussaid and Ahmed Toumanari. Web application attacks detection: A survey and classification. *International Journal of Computer Applications*, 103(12), 2014.

[11] Milton Friedman. *Free markets for free men*. Graduate School of Business, University of Chicago, 1974.

[12] Jeremiah Grossman. Cross-site request forgery. 2007.

[13] Peter Gutmann. Secure deletion of data from magnetic and solid-state memory. In *Proceedings of the Sixth USENIX Security Symposium, San Jose, CA*, volume 14, pages 77–89, 1996.

[14] Philipp Holzinger, Stefan Triller, Alexandre Bartel, and Eric Bodden. An in-depth study of more than ten years of java exploitation. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 779–790, New York, NY, USA, 2016. ACM.

[15] Amit Klein. Dom based cross site scripting or xss of the third kind. *http://www.webappsec.org/projects/articles/071105.shtml*, 2005.

[16] B. W. Lampson. Computer security in the real world. *Computer*, 37(6):37–46, June 2004.

[17] Felix Lau, Stuart H Rubin, Michael H Smith, and Ljiljana Trajkovic. Distributed denial of service attacks. In *Systems, Man, and Cybernetics, 2000 IEEE International Conference on*, volume 3, pages 2275–2280. IEEE, 2000.

[18] Henry Lieberman and Christopher Fry. Will software ever work? *Communications of the ACM*, 44(3):122–124, 2001.

[19] Net Losses. Estimating the global cost of cybercrime. *McAfee, Centre for Strategic & International Studies*, 2014.

[20] Dimitris Mitropoulos, Konstantinos Stroggylos, Diomidis Spinellis, and Angelos D. Keromytis. How to train your browser: Preventing xss attacks using contextual script fingerprints. *ACM Trans. Priv. Secur.*, 19(1):2:1–2:31, July 2016.

[21] David Nichols and Michael Twidale. The usability of open source software. *First Monday*, 8(1), 2003.

[22] OWASP. Owasp periodic table of vulnerabilities - insufficient authentication/authorization. https://www.owasp.org/index.php/OWASP_Periodic_Table_of_Vulnerabilities_-_Insufficient_Authentication/Authorization, 2013.

[23] OWASP. Testing for reflected cross site scripting. https://www.owasp.org/index.php/Testing_for_Reflected_Cross_site_scripting_(OTG-INPVAL-001), 2015.

[24] OWASP. Cross-site request forgery (csrf). https://www.owasp.org/index.php/Cross-site_Request_Forgery(CSRF), 2017.

[25] OWASP. Cross-site request forgery (csrf) prevention cheat sheet. https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)_Prevention_Cheat_Sheet, 2017.

[26] OWASP. Cross-site scripting (xss). https://www.owasp.org/index.php/Cross-site_Scripting_%28XSS%29, 2017.

[27] OWASP. Owasp the free and open software security community. https://www.owasp.org/index.php/Main_Pagel, 2017.

[28] OWASP. Owasp top 10 - 2017. Technical report, OWASP, 2017.

[29] OWASP. Sql injection. https://www.owasp.org/index.php/SQL_Injection, 2017.

[30] OWASP. Sql injection prevention cheat sheet. https://www.owasp.org/index.php/SQL_Injection_Prevention_Cheat_Sheet, 2017.

[31] OWASP. Xss (cross site scripting) prevention cheat sheet. https://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet, 2017.

[32] Paul Ralph. The illusion of requirements in software development. *Requirements Engineering*, 18(3):293–296, 2013.

[33] David Ross. Happy 10th birthday cross-site scripting, 2009.

[34] D Rumsfeld. As we know, there are known knowns. there are things we know we know. we also know there are known unknowns. that is to say we know there are some things we do not know. but there are also unknown unknowns, the ones we dont know we dont know. *Department of Defense news briefing*, 2002.

[35] Deborah Russell and GT Gangemi. *Computer security basics*. " O'Reilly Media, Inc.", 1991.

[36] Ryan Russell. *Hack proofing your network*. Syngress, 2000.

[37] Kevin Spett and Cross-Site Scripting. Are your web applications vulnerable. *SPI Labs whitepaper*, 2005.

[38] Martin R Stytz and James A Whittaker. Caution: This product contains security code. *IEEE Security & Privacy*, 99(5):86–88, 2003.

[39] Symantec. Internet security threat report. *"Symantec Corporation"*, 21, 2016.

[40] Symantec. Internet security threat report. *"Symantec Corporation"*, 21, 2017.

[41] Tommi Unruh, Bhargava Shastry, Malte Skoruppa, Federico Maggi, Konrad Rieck, Jean-Pierre Seifert, and Fabian Yamaguchi. Leveraging flawed tutorials for seeding large-scale web vulnerability discovery. *arXiv preprint arXiv:1704.02786*, 2017.

[42] Philipp Vogt, Florian Nentwich, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Cross site scripting prevention with dynamic data tainting and static analysis. In *NDSS*, volume 2007, page 12, 2007.

[43] Joel Weinberger, Prateek Saxena, Devdatta Akhawe, Matthew Finifter, Richard Shin, and Dawn Song. A systematic analysis of xss sanitization in web application frameworks. In *European Symposium on Research in Computer Security*, pages 150–171. Springer, 2011.

[44] Candid Wueest. The continued rise of ddos attacks. *Security Response by Symantec version 1.0–Oct*, 21, 2014.

[45] Mingyi Zhao, Jens Grossklags, and Peng Liu. An empirical study of web vulnerability discovery ecosystems. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 1105–1117. ACM, 2015.