The Developer is the Enemy

Glenn Wurster
Carleton Computer Security Lab
School of Computer Science
Carleton University, Canada
gwurster@scs.carleton.ca

P. C. van Oorschot
Carleton Computer Security Lab
School of Computer Science
Carleton University, Canada
paulv@scs.carleton.ca

ABSTRACT

We argue that application developers, while often viewed as allies in the effort to create software with fewer security vulnerabilities, are not reliable allies. They have varying skill sets which often do not include security. Moreover, we argue that it is inefficient and unrealistic to expect to be able to successfully teach all of the world's population of software developers to be security experts. We suggest more efficient and effective alternatives, focusing on those developers who produce core functionality used by other developers (e.g. those who develop popular APIs – *Application Programming Interfaces*). We discuss the benefits of designing APIs which can be easily used in a secure fashion to encourage security. We also introduce two straw-man proposals which integrate security into the work-flow of an application developer. Data tagging and unsuppressible warnings provide the basis for further work where the most natural use (path of least resistance) results in secure code. We believe there are benefits to co-opting developers into programming securely.

Categories and Subject Descriptors

D.4.6 [**Software**]: Security and Protection; D.2.3 [**Software**]: Coding Tools and Techniques; D.2.6 [**Software**]: Programming Environments

General Terms

Human Factors, Security

Keywords

software developers, human factors, software security, usability, education, development tools, persuasion

1. INTRODUCTION AND OVERVIEW

According to Adams et al. [1], many security policies are enforced on a need-to-know basis. This need-to-know mentality seems historically to have been based on the idea that increased knowledge of security mechanisms and threats increases the potential for information leaks. The authors argue that this need-to-know mentality results in a situation where users are less motivated to work securely. In a related position, Vidyaraman et al. [46] argued that it can be beneficial to security to consider users as the enemy, in that their actions directly influence system security and they often perform tasks that actively reduce security.

Application developers are currently treated different than users. Often, API (*Application Programming Interface*) developers provide functionality and application developers use this functionality to create applications. The security community relies on application developers to be knowledgeable and to understand how to use each API securely. In effect, we rely on all application developers to be security experts. In recent years, it has been widely acknowledged that software developers do not by any means have sufficient security expertise to make this model work [47, 6]. Consequently, some major players in industry have mounted substantial efforts towards increasing the security knowledge of general developers [26]. We argue that not only is relying on all developers individually to code securely doomed to failure, but that extensive training (even if it were possible) to give *all* software developers detailed security expertise is not the right approach. We suggest additional focus on providing development environments where even application developers without security expertise are less likely to make security errors; as developer skill sets become increasingly customized, requiring all developers to have security expertise as a core competency is too heavy a tax to pay.

It has often been said that *complexity is the enemy of security*. This complexity is present at the user interface level of software, in programming libraries and tools available to the developer of an application, as well as in system code.

The modern developer no longer builds applications from scratch. Instead, most developers essentially glue different libraries together to perform a task. Different developers are responsible for different parts of the resulting application. Given this situation, it is unreasonable to assume or require that all developers will be properly educated and proficient in security (e.g. graphic artists are unlikely to be well versed in web server application security issues, and it's not clear why they should be, as we don't require security experts to be artists). While it is generally accepted that "more user training" is not a viable solution to some security problems, apparently many in the security community continue to believe that developer education will solve the problem. In fact, we have been placing more burden on the

NSPW'08, September 22–25, 2008, Lake Tahoe, California, USA.

developer to create usable security mechanisms as a result of continuing research into encouraging secure user behaviour [17, 22, 21, 31]. Conservatively considering developers as the enemy motivates us to take the security of the system out of their hands. In order to do this, we argue the best approach is to develop and enforce technical solutions instead of assuming that developers can be educated to *do the right thing* (and actually do it).

Furthermore, not all libraries are well documented in their proper use, and security caveats known to experts remain little-known to most application developers. This is exasperated by the fact that library developers are often in different groups, companies, or countries than the application developers who use their libraries.

Developers both directly and indirectly affect the security of systems. While we know that developers can affect the security of a system in many different ways (e.g. allowing buffer overflows), to date, the focus has been on giving developers *more* options and tools in order to improve security. On a related note, because developers are task oriented, they are often the most vocal in requesting additional functionality which allows them to complete their task more easily or faster (even if this sometimes means reduced security [16]). A recent example of this is the numerous requests for a loosening of the same-origin policy [35, 49], or the expanding banking APIs [2]. Apparently many developers requesting additional functionality are not fully aware of the negative security implications of their requests.

In the following sections, we examine the issue with relying on application developers to operate securely and the problems that causes. We explore the problem from the perspective of the security of programming APIs and tools, pointing out the problems of leaving security to application developers (who may not be experts in security). We believe that the current status quo of relying on all developers for security results in a greater number of application vulnerabilities. We consider what might be done to relieve some of the burden currently placed on application developers.

The remainder of this paper is organized as follows. Section 2 reviews some current approaches at improving security in light of viewing the developer as the enemy. Section 3 examines the problems and advantages of incorporating usable security into developers' libraries and tools. Section 4 discusses how developers unfamiliar with security end up contributing to the problem, and influencing security negatively. Section 5 discusses how to co-opt developers into using security tools and libraries. Section 6 presents two new straw-man proposals which encourage secure application development. Section 7 discusses related work. We conclude in Section 8.

2. REVIEWING SOME APPROACHES TO IMPROVING APPLICATION SECURITY

While we are not the first to propose securing developer tools, we believe proposed solutions fail to address the entire problem.

In the security community, attackers are always regarded as the enemy. For each new technology proposed, the critical question of how it can be bypassed is also asked. With developers, the same has not traditionally been true. New technologies are introduced and it is assumed the developer will correctly use the technology. Operating as if the developer is the enemy, we must examine ways in which the developer can bypass or misuse the technology. When the developer is presented with an option, we can not assume they will choose the more secure alternative. We now comment on several existing technologies in light of viewing developers as the enemy.

Type-Safe Languages. While these languages exist, nothing forces the developer to use them (as discussed in Section 4.2). Even though type-safe languages prevent some classes of programming errors, other errors can still be made by the developer (and even strongly-typed languages may contain subtle security issues [29]). In order to permanently eradicate certain classes of programming errors (e.g. buffer overflows), type-safe languages would need to be mandated in all environments.

Security Analysis Tools. As discussed in Section 3.3, there is no assurance that a developer will always run a particular security tool or understand the results. In treating the developer as the enemy, we must somehow persuade them to use available security analysis tools. We must also ensure the tools are designed such that the security expertise required to use them is minimal.

Secure Libraries. As discussed in Section 4.2, developers are not obliged to use one library (or API) over another. In treating developers as the enemy, we would need to prevent them from using insecure libraries. We believe, however, that preventing the distribution or use of insecure libraries is akin to blocking the distribution of illegal copies of copyrighted work – a near-impossible task. We therefore concede that benefits in this area are most likely to come from ensuring secure APIs are also the easiest to use (as discussed in Section 3.4).

Auto-Escaping Data. Auto-escaping input data [38] has the potential to fix many security vulnerabilities (including XSS and SQL injection). Input data to an application is automatically escaped by the interpreter in an attempt to protect against common vulnerabilities (normally associated with output data). Commonly escaped characters include quotes in an attempt to protect against SQL injection attacks. Unfortunately, it does not always work. The input data is escaped before it is known what the data will be used for. Input data can be used for any number of different purposes, each of which requires a different form of escaping in order to sanitize the data properly. With auto-escaping enabled, the application is still forced to unescape the input and then re-escape it correctly depending on the output data format.[1] Because of this inability to escape properly, many developers write code to remove escaping before input is processed by the rest of the application – negating the benefits of auto-escaping. Only output APIs have a reasonable possibility of being able to correctly escape untrusted data.

3. SECURITY AND USABILITY OF API'S AND DEVELOPERS' TOOLS

Traditionally, usability and security research has focused on software applications as used by end users. How this is impacted indirectly by software developers, and the APIs

[1]Because of its inability to determine the proper output escaping format, `magic_quotes_gpc` functionality (which auto-escapes input) has been removed from PHP version 6 (along with `register_globals` as discussed in Section 3.3).

they use, has received much less attention. This relationship can be seen by examining Figure 1. In modern development environments, the role of the application developer is often to combine functionality from many different APIs (and associated libraries).

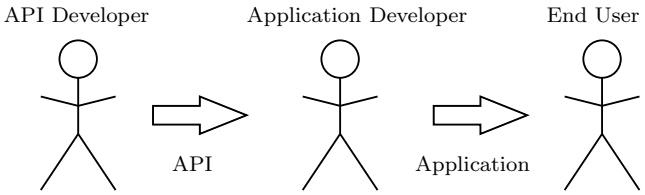

Figure 1: Creation and flow of developer code

We note that developers, as end-users of development tools, can benefit from usable security research into development tools, as they affect the security of resulting applications. We also note that even though a developer is generally an expert computer user, this does not mean that they can be relied upon to use APIs in a secure way. Additionally, we do not believe developers can reasonably be relied upon to "correctly" use security tools with poor user interfaces; or in fact to willingly use them at all.

Our view of application development in Figure 1 provides a finer granularity perspective on what impacts the security of an application. If the libraries used by the application developer are not secure, then the resulting application is also unlikely to be secure[2] (e.g. as a well-known example, the `gets` family of C library functions leads to many vulnerable programs). This provides an opportunity: we believe security vulnerabilities can be reduced by focusing on a relatively small number of widely used programming libraries (as opposed to each individual application). Some simple security guidelines already exist for API development (e.g. always take a length parameter with a buffer, security sensitive data structures should have a version number, etc.). We encourage study on how more subtle API changes affect the security of subsequent applications (including changes affecting aspects of the API covered by our broader definition in Section 3.1). We do not believe shifting responsibility for verifying program security from the developer to end-user to be a wise solution (even through mechanisms such as approval dialog boxes).

3.1 What Constitutes an API

We use the term "API" in this paper a bit differently than many others. The API is traditionally defined as only the interface for accessing programming functionality – a collection of function prototypes and data types. We find it beneficial to view an API as more than the set of function prototypes. In this paper, we expand the term "API" to cover several additional attributes. In our use of the term, the API description also takes into account all calling restrictions.

From a security perspective, all APIs implement some form of access control to a resource (be it fully open or otherwise). Access control policy is enforced through what functionality the API exposes (e.g. not providing a write API function to prevent file modifications) and the exceptions thrown when illegal actions are attempted (e.g. writing to a file you don't have access to). There also may be restrictions on the order that API functions can be called (e.g. writing to a file which is not open may violate the calling restrictions, resulting in an error). The code implementing the functions exposed in an API enforces all these order and access control calling restrictions. In order to accomplish a task, applications using an API must not only call the functions correctly, but also abide by the additional calling restrictions. As a concrete example, we would consider the same-origin policy [40] intrinsically tied to the API which a web browser makes available to JavaScript code, even though it is not related to any particular function prototype or argument. Capability based systems have long restricted access to sensitive objects based on context [15].

From a security perspective, violations to the calling restrictions should always be flagged as errors instead of begin allowed (leading to unexpected, undefined, or unintended consequences). This is especially true when the developer cannot be relied upon to use the API correctly, as is the case when we consider them to be the enemy.

3.2 Security, Usability and End-Security of APIs

The insight that APIs affect application security leads us to a new matrix of API design factors. The traditional view of usability and security (see Figure 2) dictates that an interface can be usable (A), secure (B), both (C), or neither (D). We say that an API is secure if the functionality can not be exploited to perform some undesired task [9]. We say that an API is usable if the developer can easily use it to accomplish a task [32]. Traditionally, we have attempted to make APIs which are usable. More recently, we have started to focus on APIs which are secure. This is still not the full picture, however. There is a third category which has been relatively unexplored (to our knowledge). We note with emphasis that design choices made when developing an API can also influence the security of applications which are built using that API. Similar to cryptography, there are algorithms which are secure but may be commonly used incorrectly in protocols (leading to vulnerabilities). This expanded view of how the API affects application security is shown in Figure 3 (where a good API would fall near E). We say that one API promotes security better than a second API if applications employing the first API are more likely to be free of vulnerabilities than those employing the second. While "security promoting" characteristics remain an open research area, we discuss one proposal in Section 6.1.

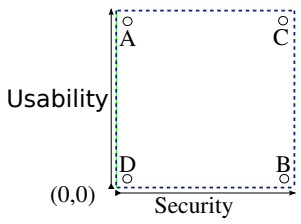

Figure 2: Usability and security box

[2]A security expert may be able to select and use only the secure pieces of an insecure library, but as we mention earlier, we should not assume that all developers are also security experts.

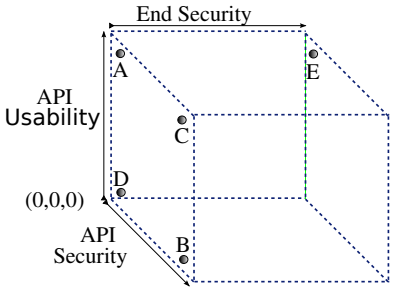

Figure 3: API usability, API security, and end security (i.e. security of resulting application) cube. A fourth dimension could be added, usable security of the resulting application.

3.3 Support Tools

While support tools such as static analysis checkers and program analyzers have the capability to detect some classes of programming errors [28], it seems clear that such tools cannot hope to protect against all programming-induced vulnerabilities. Any support tool which the developer must independently run is also not likely to be run by all developers. Furthermore, many of these support tools still require manual intervention on the part of the developer, who must also be able to understand the security error reported by the tool and respond properly to it. Another threat is from developers who are unaware of new security technologies, or who ignore them. These are developers we cannot influence easily.

There is an interesting lesson to be learnt from the example of attempting to protect against buffer overflow attacks. Many tools aimed at developers were designed to assist in detection of buffer overflows [51], but few of these (if any) enjoy widespread deployment. In contrast, library changes and minor API tweaks to the major operating systems (e.g. no-execute memory protection [33, 10] and address layout randomization [7]) were deployed and had a much larger impact on security. We believe that this example provides a strong case for focus on designing and deploying solutions that do not require ordinary developers to become security experts.[3] The libraries and API now default to a more secure state. With no-execute memory protection, some old applications such as just-in-time compilers had to be updated to request executable data pages. We perceive security mechanisms which are *invisible* to the application developer[4] as having the greatest effect.

Another example of changing an API to increase security is the removing of `register_globals` in PHP. This functionality causes all user-supplied input to be assigned to global variables in a PHP script. It was realized that this functionality lead to a large number of security vulnerabilities and hence it has been removed from PHP 6 [37]. Application developers can no longer use `register_globals`. Both these examples have required old programs to be updated.

[3]In fact, it also presents an argument against removing security protections perceived as outdated – segmented addressing also would have reduced the number of exploitable buffer overflows.

[4]While the security mechanism can not be invisible to all developers, most developers will not need to concern themselves with the policy in order to operate within its constraints.

3.4 Working within a security mechanism

If we are to attempt to create, or encourage the creation of, programming libraries (and associated APIs) which result in applications with fewer vulnerabilities, we must also ensure that developers can easily use the resulting APIs. While this remains an open research problem, it is worth noting a few observations in this area.

1. *User interface of APIs.* As application developers become experienced with an advanced API, they often find more efficient or secure ways of using its functionality. How to design a function and document its API so as to reduce the time it takes to become proficient with its use is an open usability and security question we pose. We have learnt from cryptography that providing cryptographic algorithms alone does not ensure security [52, 8]; using these algorithms securely is more difficult. The same is true for many software libraries.

2. *Developer-friendly security mechanisms.* As discussed in Section 1, many of the security mechanisms proposed to increase end-user security place an additional burden on the application developer (e.g. PKI certificates are not easy for developers to work with [45]). Another open research area is how to design and deploy new security mechanisms which do not increase the security expertise required by generic application developers – that is, which are more usable by developers.

3. *Integrated vs. opt-in security tools.* Many security-related development tools are "opt-in." The developer must explicitly run or enable the tool. It seems clear that standalone tools will never enjoy the same deployment and use as integrated security solutions, just as opt-in security functionality for end-users has traditionally yielded underwhelming results. Security solutions embedded into the libraries used by developers will enjoy the broadest distribution.

4. HOW DEVELOPERS INDIRECTLY INFLUENCE SOFTWARE SECURITY

Developers end up indirectly influencing the security of a system in many ways. We now discuss four classes of examples.

4.1 Feature Requests which Negatively Impact Security

Application developers are task oriented and often request additional functionality from API developers. This sometimes negatively affects security. One example is the request to loosen the same-origin policy [35, 49], which currently prevents JavaScript on one domain from being able to access properties on content from another domain [40, 41]. This helps isolate web client applications from each other.[5] This separation is currently based on domain name. If a server makes available a web client application at one domain, it can be reasonably assured that clients interacting with the server are not scripts associated with a different domain; scripts associated with a remote domain are limited to

[5]We define web client applications as those applications running in the web browser on a client, and web server applications as applications which run on the server and make available content to the web browser.

simple operations [30]. This design, however, restricts developers who would like to involve multiple unrelated domains in the same web client application. Despite the negative security implications, developers are requesting [49, 35] that web client applications be allowed to interact with domains other than the origin.

4.2 Choice of Libraries and Languages

A second way in which developers end up strongly influencing security is when there are multiple libraries available that perform the same operation. Developers will, in general, choose the library which is easiest to use. If this happens to be insecure, then an insecure library may become the most often used (and hence standard) library. Ensuring that the libraries considered to be the most secure (among alternatives) have usable APIs is important if we wish to encourage use of secure libraries. If a secure but relatively unusable implementation of functionality exists, developers are likely to create another implementation which is more usable but perhaps less secure. In this regard, developer preferences can be a powerful influence on software security.

The same can be said of programming languages. When writing a program, many developers are likely to choose the language with which they are most familiar, regardless of the security of the language.

4.3 Perceived Trade-offs

One common argument against security is the perceived performance penalty. Developers may intentionally ignore security in an effort to have their code run more quickly. Furthermore, developers will disable or ignore compile-time tools designed to assist in creating secure code in order to meet production deadlines. Developers often believe that their code is not security critical. With the interconnected nature of today's applications, however, a vulnerability in a single application can have an effect on the security of the entire system [18]. Such developers are choosing to trade security in favour of other aspects of the software.

Related to security trade-offs is the the fact that developers are rarely measured on the security of their code. Focusing on code security often does not lead to any short-term reward (and developers, like others, are poor at evaluating long-term consequences [4]).

4.4 Testing is not a first-choice job

A third way in which developers end up dictating security has to do with the way they create software. When faced with implementing a piece of software, developers typically break the task up into a set of features (or sub-tasks), each of which may be further broken down into manageable tasks which can be implemented sequentially and tested. This practise happens regardless of the design process.

This development style can affect the security of a product in several ways:

1. Developers, like other creative individuals, thrive on novelty, and implementing a new feature provides a greater sense of accomplishment than testing it. Given the choice of implementing a feature or running vulnerability analysis tools, many developers prefer to implement the feature.

2. Most developers concentrate best on a single task at a time. In fact, many source control systems implicitly assume that all files edited are part of the same feature when they are checked in. One needs to create multiple distinct copies of the source in order to work on different features – a time consuming process in large projects.[6] Often developers may realize in the course of implementing a feature that they need to go back and check other areas of the code to ensure that they did not make a mistake. Usually, these checks will be put off until the current task is done. By putting off checking other code until the current feature is done, developers run the risk of forgetting to do testing, some of which may be security related.

5. EMBEDDING SOLUTIONS

In keeping with the rest of the paper, we realize that developing any new security tool in isolation is insufficient. Any technology developed must be embedded into the tools the developer actually uses. We see three methods of persuading developers to use security technologies.

1. Eliminate aspects of security tools which incline developers to avoid the tools. This involves ensuring that the tools developed are user-friendly. During the design process, usability must be considered as critical in designing tools. This goes against conventional reasoning that usability is less critical in designing tools only used by experts, but is not inconsistent with reasoning in the fields of human factors [39].

2. Provide the developer with rewards or incentives for coding securely (related to Section 4.3) or for using security tools. Rewards can be company internal (e.g. financial incentives by management) or external (e.g. documenting the number of warnings still existing in a shipped product). For those companies where management does not take security seriously, the rewards or incentives would have to be external.

3. Mandate the use of specific tools or libraries. This works well for large companies with leverage (e.g. forcing third-party developers to use analysis tools, specific languages, specific compiler flags, and other elements). Mandates can also come from management within a company (e.g., forcing all developers to run a static analysis tool). Clients, system vendors, and others can also mandate the use of security tools.

While each of these approaches may work in a specific circumstance, the best solution likely involves some mix of all three. The hope is that contributing positively to usable security solutions leads to developers being more enthused, building customer loyalty instead of resentment.

6. STRAW-MAN PROPOSALS TO INCREASE APPLICATION SECURITY

Because not all developers have security as a primary skill, we believe there is a benefit in studying methods other than developer education to improve application security. In this section, we examine two straw-man proposals designed to incorporate security into the work-flow of developers. These proposals are designed to assist developers in creating secure applications. We recognize that these straw-man proposals must still be included into the developer work-flow through some method (see Section 5).

[6]There are exceptions to this, e.g., the recent source control system GIT [23] allows each developer to maintain their own source-control tree containing patches which have not been checked into the main repository.

6.1 Data Tagging

A good example of specialization of developer skill sets is the use of template engines as they relate to modern web development. Template engines such as Smarty [43] allow program functionality to be separated from web site visual layout. They allow core developers to concentrate on back-end functionality without worrying about how this content is displayed. Furthermore, graphic designers are able to customize the template to increase the visual aesthetic of the web site without touching core functionality. The template language provides a mechanism which the graphic designer can use to access variables which the back-end explicitly makes available. The graphic designer can be expected to be artistic, without being a proficient developer.

The problem with this approach is that some security vulnerabilities are directly related to the display of data exported by the back-end. User-submitted data exported to the graphic designer by the back-end must be properly escaped to prevent cross-site scripting (XSS) vulnerabilities [24]. Escaping data correctly requires knowing both the context in which the data is displayed, and the markup language being used – both of which, with current tools, are determined at display time by the graphic artist. This places the security of websites in the hands of graphic artists, requiring them to be security experts – to properly escape all data that originates from user input. Because determining what input is from the user is a task best done in the core functionality of the web server application, we explore a method of making this information available in the API used by graphic artists in the template engine language. Our proposal helps facilitate coding securely (for all developers, including graphic artists).

A Tagging Proposal. To further motivate our first proposal, consider an API which returns data to a developer (a more general case of the template engine discussion above). Making this data explicitly available to an application may result in it being used insecurely (e.g. to compromise privacy). One effective defence which is well known, but seldom used (for various reasons) is to encapsulate the data in an object which provides limited access methods, with raw data never being available to the application developer. Because of the increased number of functions required to support various uses of the data, the size of the API is large (which tends to discourage use somewhat). Taint checking has also been proposed to combat the problems of using untrusted data returned by an API [34, 48]. However, this requires security-aware developers, and only provides an indication that the data is potentially malicious, not why, what it can be used for, or how to neutralize it.

Our proposal is to extend the idea of tainting to *data tagging*. Instead of tainting data (signalling that it may be bad for *some* reason), the API developer tags the data with various labels, indicating why the data requires special care (e.g. it is personal information, is from an untrusted source, was set by the user, or has not been verified as correct). In the context of template engines, data which needs to be properly escaped may be tagged with an `escapehtml` tag. Similar to tainting, a warning or error would be displayed if the developer attempts to output the data without escaping it (using available escaping functions). Multiple tags may be assigned with any piece of data. We believe that data tagging would provide a convenient method for the API developer to allow the application developer to correctly use data returned by the API, with improved usability (usable security), as the application developer is informed *why* the data is tagged. Our proposal is best suited for high-level programming languages.

Tagging becomes even more effective when tags are maintained across application boundaries (by the applications themselves, not by the system [11, 25]). If data inserted into a database retained the labels with which it had been tagged, stored XSS vulnerabilities [14] would be much easier to find by web developers and security analysts. Tagging can also be used to create an alternative to auto-escaping (Section 2), allowing proper escaping of tagged data at output time based on the destination of the data. Tagging allows input and output APIs to jointly protect application data.

6.2 Unsuppressible Warnings

Related to variable tagging is the notion of warning and error messages generated during the compiling, translating, or running of an application (in both interpreted and compiled languages). Many warnings are suppressed by the application developer in an effort to decrease development time, even though some may indicate security errors (e.g. the PHP uninitialized variable warning indicates a variable which could be set by an attacker when running with `register_globals` [37]).

We propose the following. To ensure that developers always address warnings and errors in their application code, as a general principle developer tools should be designed such that these warnings are unsuppressible. If the only ways of suppressing warnings are to fix or document them, developers should be much more motivated to ensure that their code does not generate warnings. Addressing warnings then becomes part of accomplishing the developer's primary task. The developer is forced to examine every warning and take corrective actions as part of completing their task. In effect, we propose turning warnings into errors (as can already be done with GCC by passing the `-Wall` and `-Werror` options). Despite no suitable references in the academic literature to our knowledge, several major projects already enforce a no-warning policy (including the Xen hypervisor). To reach the greatest number of developers, this change would be a mandatory setting beyond the developers control (unlike current GCC). With unsuppressible warnings, we can ensure going forward that newly detected security errors will not be ignored by developers. During the transition to unsuppressible warnings, it is inevitable that errors will appear outside the developer's code (i.e., in components designed by others). Encouraging developers to submit bugs/patches against such warnings would help improve the entire software ecosystem.

7. RELATED WORK

Our work is similar to that of Solworth at NSPW 2007 [44], which concentrated on how to successfully deprecate APIs, providing new APIs using some of the methods described in Section 2. In contrast, we examine what can be further done to increase the security of applications developed using new APIs.

Data tagging is related to the notion of data labelling, which is distinct in that its goal is to protect confidentiality [5]. Data tagging allows multiple tags to be assigned to a single variable, which is in contrast to allowing a single data label. Data security policies (such as Clark-Wilson [12]) dif-

fer from data tagging fundamentally in their response to access violations. One purpose of tagging is to provide information suggesting what may need to be done in order to sanitize the data. Furthermore, the Clark-Wilson model concentrates on protecting data from the system, while tagging concentrates on protecting the system from the data. Data provenance [42] is concerned with documenting the origin of data and all transformations performed on that data. Tagging is related to identifying the source of data, but is not concerned with logging all data transformations.

Languages like Ada [27] and Java use strong type-checking stop certain classes of programming errors. Such strongly-typed languages are not a universal solution because developers normally choose a language based on functionality and ease of use [36]. Backwards compatibility and support are also issues which developers examine when choosing a language.

AEGIS [20] focuses on assisting developers in creating secure software, by trying to incorporate security into the software design process. While this approach is valid, it does not help software not designed under the AEGIS model.

Other work includes fault isolation [50] and sandboxes (in both browsers [3] and virtual machines). Both fault isolation and sandboxes attempt to restrict code which may contain bugs, but do not work to reduce the number of vulnerabilities which may be present in the software. Our approach of focusing on usable security attempts to reduce the number of vulnerabilities which exist.

Vidyaraman et al. [46] explored the position that the user was the enemy in the computer system.

Microsoft's Windows Vista contains *User Account Control* (UAC), which was partially designed to encourage developers to restrict security sensitive operations [19]. We believe that users do not need to be adversely affected in order to encourage developers.

In the banking industry, API security research has treated the developer as the enemy, but has concentrated solely on protecting secrets (e.g. encryption keys) behind the API [2]. Our research attempts to increase the security of the applications developed as well as the libraries.

Agile software development methods [13] decrease the number of security vulnerabilities, and we view them being complementary to methods proposed in this paper.

8. CONCLUDING REMARKS

In this paper, we explored the idea that developers influence software security in many ways (including negatively), and how the tools and APIs they use are a big part of this. While security and usability have been previously studied in the context of end-users, we have seen little prior focus on the problem of usable security as it relates to programming libraries, and the resulting impact on applications.

By realizing that not all developers are security experts, and by improving the security of tools and libraries the typical developers use, we have the potential to positively influence all developers to code in a more secure fashion, resulting in fewer vulnerabilities. It is possible that as a byproduct of repeated use of appropriate security tools, developers will change their coding habits in a way which improves the security of their code. In cases where software libraries are used by many different applications, the potential exists to influence the security of a large number of applications by concentrating on the security of a single library and associated API. While the security benefit to be gained by focusing on APIs is hard to quantify, we believe there are significant advantages still to be found.

Related to the role that developers play in security is the role that product testers play. While we have focused on developers in this paper, the usability of testing tools and how they influence security is a related topic which merits additional research. The possibility of using tools which automatically identify security issues during the specification phase of software development also merits additional research but is beyond the scope of this paper.

We also discussed the idea of incorporating security into the task based work-flow of developers. We described two straw-man proposals which have the potential to encourage secure software development, even amongst developers not well versed in security.

As was discussed, many security vulnerabilities are the result of poor developer choices. Our position is that removing some of these choices should help all developers perform their tasks more securely (even if they are not security experts). A main challenge in specific instances is to find a restricted set of options, or single option, which is suitable, agreeable, and useful to everyone.

Because there are a great many more developers than security experts, we encourage focus by the latter on the security of tools and libraries which many developers use. We believe it is a loosing proposition to rely too heavily on universal education and opt-in security tools if the goal is to ensure that all developers create secure applications.

Acknowledgements. We thank NSPW participants and anonymous reviewers for their comments. The first author acknowledges NSERC for funding his PGS D scholarship. The second author acknowledges NSERC for funding a Discovery Grant and his Canada Research Chair in Network and Software Security. Both authors acknowledge MITACS for partial funding.

9. REFERENCES

[1] A. Adams and M. A. Sasse. Users are not the enemy. *Communications of the ACM*, 42(12):41–46, Dec 1999.

[2] R. Anderson. *Security Engineering*, chapter 18: API Security. Wiley, 2nd edition, 2008.

[3] V. Anupam and A. Mayer. Security of web browser scripting languages: vulnerabilities, attacks, and remedies. In *Proc. 7th USENIX Security Symposium, 1998*, pages 15–28, 1998.

[4] A. Beautement, M. A. Sasse, and M. Wonham. The compliance budget: Managing security behaviour in organisations. In *Proc. 2008 Workshop on New Security Paradigms*, Sep 2008.

[5] Y. Beres and C. I. Dalton. Dynamic label binding at run-time. In *Proc. 2003 Workshop on New Security Paradigms*, pages 39–46, 2003.

[6] K. Beznosov and B. Chess. Security for the rest of us: An industry perspective on the secure-software challenge. *IEEE Software*, 25(1):10–12, Jan 2008.

[7] S. Bhatkar, D. C. DuVarney, and R. Sekar. Address obfuscation: an efficient approach to combat a board range of memory error exploits. In *Proc. 12th USENIX Security Symposium*, pages 8–23, Jul 2003.

[8] A. Bittau, M. Handley, and J. Lackey. The final nail in WEP's coffin. In *Proc. 2006 IEEE Symposium on Security and Privacy*, pages 386–400, May 2006.

[9] M. Bond and R. Anderson. API-level attacks on embedded systems. *Computer*, 34(10):67–75, Oct 2001.

[10] Z. Brown. Kernel traffic #265 for 30-Jun-2004. Web Page (viewed 29 Mar 2008), Jun 2004. `http://www.kerneltraffic.org/kernel-traffic/kt20040630_265.html#4`.

[11] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum. Understanding data lifetime via whole system simulation. In *Proc. 13th USENIX Security Symposium, 2004*, pages 321–336, Aug 2004.

[12] D. Clark and D. Wilson. A comparison of commercial and military computer security policies. In *Proc. 1987 IEEE Symposium on Security and Privacy*, pages 184–194, 1987.

[13] A. Cockburn. *Agile Software Development*. Addison Wesley, 2002.

[14] N. Daswani, C. Kern, and A. Kesavan. *Foundations of Security: What Every Programmer Needs to Know*, chapter 10: Cross-Domain Security in Web Applications. Apress, 2007.

[15] J. B. Dennis and E. C. Van Horn. Programming semantics for multiprogrammed computations. *Communications of the ACM*, 9(3):143–155, Mar 1966.

[16] A. J. DeWitt and J. Kuljis. Aligning usability and security: a usability study of Polaris. In *Proc. 2nd Symposium on Usable Privacy and Security*, pages 1–7, 2006.

[17] R. Dhamija and J. D. Tygar. The battle against phishing: Dynamic security skins. In *Proc. 2005 Symposium on Usable Privacy and Security*, pages 77–88, 2005.

[18] Security vulnerability in tetex (dvips). Web Page, 10 2002. `http://www.securityfocus.com/advisories/4567`.

[19] K. Fisher. Vista's UAC security prompt was designed to annoy you. Web Page (viewed 14 Apr 2008), Apr 2008. `http://arstechnica.com/news.ars/post/20080411-vistas-uac-security-prompt-was-designed-to-annoy-you.html`.

[20] I. Flechais, M. A. Sasse, and S. M. V. Hailes. Bringing security home: a process for developing secure and usable systems. In *Proc. 2003 Workshop on New Security Paradigms*, pages 49–57, 2003.

[21] A. Y. Fu, X. Deng, L. Wenyin, and G. Little. The methodology and an application to fight against unicode attacks. In *Proc. 2nd Symposium on Usable Privacy and Security*, pages 91–101, 2006.

[22] S. Gaw and E. W. Felten. Password management strategies for online accounts. In *Proc. 2nd Symposium on Usable Privacy and Security*, pages 44–55, 2006.

[23] GIT user's manual (for version 1.5.3 or newer). Web Page (viewed 29 Mar 2008), Mar 2008. `http://www.kernel.org/pub/software/scm/git/docs/user-manual.html`.

[24] HTML code injection and cross-site scripting: Understanding the cause and effect of CSS (XSS) vulnerabilities. Web Page (viewed 4 Apr 2008), Apr 2008. `http://www.technicalinfo.net/papers/CSS.html`.

[25] A. Ho, M. Fatterman, C. Clark, A. Warfield, and S. Hand. Practical taint-based protection using demand emulation. In *Proc. 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, pages 29–41, 2006.

[26] M. Howard and S. Lipner. Inside the windows security push. *IEEE Security and Privacy*, 1(1):57–61, Jan 2003.

[27] J. D. Ichbiah, B. Krieg-Brueckner, B. A. Wichmann, J. G. Barnes, O. Roubine, and J.-C. Heliard. Rationale for the design of the Ada programming language. *ACM SIGPLAN Notices*, 14(6b):1–261, Jun 1979.

[28] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities (short paper). In *Proc. 2006 IEEE Symposium on Security and Privacy*, May 2006.

[29] C. Lai. Java insecurity: Accounting for subtleties that can compromise code. *IEEE Software*, 25(1):13–19, Jan 2008.

[30] V. T. Lam, S. Antonatos, P. Akritidis, and K. G. Anagnostakis. Puppetnets: misusing web browsers as a distributed attack infrastructure. In *CCS '06: Proceedings of the 13th ACM Conference on Computer and Communications Security*, pages 221–234, New York, NY, USA, 2006. ACM Press.

[31] E. Lieberman and R. C. Miller. Facemail: showing faces of recipients to prevent misdirected email. In *Proc. 3rd Symposium on Usable Privacy and Security*, pages 122–131, 2007.

[32] S. McLellan, A. Roesler, J. Tempest, and C. Spinuzzi. Building more usable APIs. *Software, IEEE*, 15(3):78–86, May 1998.

[33] Microsoft Corporation. A detailed description of the data execution prevention (DEP) feature in Windows XP Service Pack 2, Windows XP Tablet PC Edition 2005, and Windows Server 2003. Technical report, Microsoft Corporation, Sep 2006. `http://support.microsoft.com/kb/875352`.

[34] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans. Automatically hardening web applications using precise tainting. In *Proc. 20th IFIP International Information Security Conference*, Jun 2005.

[35] F. Nimphius. Application security in AJAX. Web Page (viewed 28 Mar 2008), Oct 2007. `http://ajax.sys-con.com/read/436281.htm`.

[36] J. Ousterhout. Scripting: higher level programming for the 21st century. *Computer*, 31(3):23–30, Mar 1998.

[37] *PHP: Using Register Globals*, Nov 2007. `http://www.php.net/manual/en/security.globals.php`.

[38] *PHP: What are Magic Quotes*, Nov 2007. `http://www.php.net/manual/en/security.magicquotes.php`.

[39] J. Preece, D. Benyon, G. Davies, and L. Keller. *A Guide to Usability: Human Factors in Computing*. Addison Wesley, 1993.

[40] J. Ruderman. The same origin policy. Technical report, Mozilla, 2001. `http://www.mozilla.org/projects/security/components/same-origin.html`.

[41] J. Schuh. Same-origin policy part 1: Why we're stuck with things like XSS and XSRF/CSRF. Web Page,

Feb 2007. http://taossa.com/index.php/2007/02/08/same-origin-policy/.

[42] Y. L. Simmhan, B. Plate, and D. Gannon. A survey of data provenance in e-science. *ACM SIGMOD*, 34(3):31–36, 2005.

[43] Smarty: Template engine. Web Page (viewed 28 Mar 2008), Feb 2008. http://smarty.php.net.

[44] J. A. Solworth. Robustly secure computer systems: A new security paradigm of system discontinuity. In *Proc. 2007 New Security Paradigms Workshop*, 2007.

[45] T. Straub. *Usability Challenges of PKI*. PhD thesis, Technische Universitt Darmstadt Universitts- und Landesbibliothek, 2006.

[46] S.Vidyaraman, M.Chandrasekaran, and S.Upadhyaya. Position: The user is the enemy. In *Proc. 2007 New Security Paradigms Workshop*, 2007.

[47] I. A. Tondel, M. G. Jaatun, and P. H. Meland. Security requirements for the rest of us: A survey. *IEEE Software*, 25(1):20–27, Jan 2008.

[48] P. Vogt, F. Nentwich, N. Jovanovic, C. Kruegel, E. Kirda, and G. Vigna. Cross-site scripting prevention with dynamic data tainting and static analysis. In *Proc. 2007 Network and Distributed System Security Symposium*, pages 67–78, Feb 2007.

[49] Access control for cross-site requests. Technical report, W3C, Feb 2008. http://www.w3.org/TR/2008/WD-access-control-20080214/.

[50] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. *SIGOPS Operating System Review*, 27(5):203–216, 1993.

[51] J. Wilander and M. Kamkar. A comparison of publicly available tools for dynamic buffer overflow prevention. In *Proc. 2003 Network & Distributed System Security Symposium*, pages 149–162, 2003.

[52] H. Wu. The misuse of RC4 in Microsoft Word and Excel. In *Cryptology ePrint Archive: Listing for 2005*. 2005. http://eprint.iacr.org/2005/007.pdf.