Insecure Context Switching: Inoculating regular expressions for survivability

Will Drewry and Tavis Ormandy
Google, Inc.
{*wad,taviso*}*@google.com*

Abstract

For most computer end–users, web browsers and Internet services act as the providers and protectors of their personal information, from bank accounts to personal correspondence. These systems are critical to users' continued lifestyles but often show no evidence of survivability [45], or robustness against present and future attacks. Software defects, considered the largest risk to survivability [45], are quite prevalent in consumer products and Web service software components [12]. Recent widespread security issues [20] [19] serve to emphasize this fact and show a lack investment in survivability engineering practices [22] [23] [50] [53] that may have mitigated the risk.

Common software components that comprise industry software, commercial or free, were authored and deployed with functional isolation in mind. Despite original intent, many of these components are migrating in to Internet–connected systems. The context switch from functional isolation to extreme connectivity changes the threat environment of these components dramatically [10] [53]. Most software that has undergone this sort of insecure context switch has received very little security attention. This paper briefly surveys recent examples of these sorts of context switches. In particular, we focus on the survivability and inoculation [31] of regular expression engine implementations in connected environments. Through the course of this research, a number of critical vulnerabilities were uncovered that traverse operating systems and applications including Adobe Flash, Apple Safari, Perl, GnuPG, and ICU.

1 Introduction

The existence of software defects are attributed to many causes, ranging from expected human error to poor developer education [42] [43] [96]. In the past decade, the evolution of software use and analysis were added as complicating factors. Survivability engineering [45] focuses on these factors and the of application risk mitigation tactics at all levels, especially technical ones, to aid in fortifying software as it ages [22] [23] [50] [53]. Even though there has been extensive research into survivability engineering, very few of the guidelines [50] [53], or even other known good security practices, appear commonplace in industry software, and the general state of software security may still be as Blakley described nearly twelve years ago [10]. A basic review of software defect occurrence rates, as a contraindicator of survivability, reinforces the bleak outlook on the current environment [12] [69] [68].

Survivability engineering, however, is largely a counterintuitive concept in normal software development [53], especially when it comes to code reuse. Code reuse is encouraged [47] and common, as evidenced by the large number of commercial and open source software libraries and their extensive use across the software industry. Software libraries, or any reusable components, are often designed with specific usage contexts in mind. These uses may be well–defined using any number of formal, such as UML [71], or informal techniques, but rarely are the complete intentions effectively communicated to the end–developer, especially as these components age. There is no established standard for describing the expected threat environment of a software library, and even if there was, there is no mechanism to enforce the use of such a standard. While domain specific repositories aid in context–safe reuse [30], there are not many comprehensive repositories which match functional domains with security domains [60]. This leaves little option for developers except to rely on programming language–supplied features and software components selected by their advertised functionality. This behavior, when not carefully considered, leads to insecure context switching.

This paper contributes a thorough examination of the

survivability and inoculation [31] of a prime example of insecure context switching: modern regular expression [44] use. Regular expressions are an idiomatic technique for text search and manipulation with a presence in nearly every modern programming language, from C to Ruby. Their widespread acceptance has lead regular expressions to be exposed unquestioningly in potentially hostile, connected environments. This would be unsurprising, and unimportant, if regular expression engine implementation was trivial or if regular expression engines were implemented with connected environments in mind. However, the creation of robust implementations is still an open topic of research [48] [24], and regular expressions were implemented, initially, in functional isolation as a text editor feature [18].

1.1 Background

Survivability of specific software components is not the only measurement of overall system survivability. While a number of serious vulnerabilities are discussed later in the paper, the view should not be overly bleak. Many legacy, and even just poorly implemented, software components exist and are in common use, but there are inoculation, or risk–mitigating tactics, already at play at a lower–level.

Most modern operating systems deploy a number of risk–mitigating mechanisms to add to the overall robustness of the system. Many tactics are used explicitly to prevent software defects from becoming fully exploitable vulnerabilities. Some of these tactics are address–space layout randomization [70], non–executable memory pages [61] [95], and capability-enforcement systems [51] [5]. In addition, compilers, like GNU GCC, provide the addition of stack canaries [25] and position independent executable support [41]. The combined efforts of base system libraries, language compilers, and the operating system layers add to the overall survivability. While none of these techniques prevent exploitation entirely, they do increase the complexity of creating reliable exploits.

In addition to these base system extensions, there is work which provides risk–mitigation at the application layer. Systrace [63], presented in 2003, provides a mechanism for intrusion prevention aiding the overall survivability of the system. Unfortunately, system call–based policy enforcement systems, like systrace, are not supplied by default with most operating system distributions. There are also other software fault isolation, or software sandboxing, systems in existence [100], but their deployment is similarly scarce.

The lack of sandbox integration and the unenforced nature of many of the base system mitigation tactics leaves single components liable for full system survivability. Dowd's recent whitepaper [20], describes such a scenario. Flash suffered from an exploitable vulnerability, but as it was not compiled in a fashion compatible with Windows Vista's ASLR functionality, no randomization occured which allowed for reliable exploitation on Vista. This is not an isolated incident; many systems and software do not support or use these features [65] [13], and when they do, they are still not impervious to all known attack techniques, much less future developments in software security research.

1.1.1 Regular Expression History

Since the introduction of regular expressions to the QED text editor [18] and the Unix environment in the 1970s [94], regular expression use, as a means of succinctly expressing acceptable grammars, has increased continuously. They have even garnered an accepted place in many IETF standards. A simple review of IETF standards, or RFCs, shows a continued reliance on regular expressions. This is likely due, in large part, to the release of a public domain, regular expression engine by Henry Spencer in 1986 [72] [73].

Beginning around 1990, regular expressions moved from a command–line environment and text editor feature to an expected piece of functionality in most programming languages. Starting with Tcl, EMACS/Lisp [49], and Perl [92], this trend [11] continues into modernity with PHP, Python, Ruby, Javascript, Actionscript, C#, Java, and so on. In spite of the continued inclusion of regular expressions as core functionality, many of the engines derive their code, at least tangentially, from Spencer's 1986 release. This ancestry leads to heavily patched implementations blind to newer, or even parallel, advances in this area of research [48] [14]. Even for engines that have emancipated themselves from this legacy, regular expression implementation is not a trivial task.

1.1.2 Regular Expression Security

Past research into the security of regular expression engine implementation has been limited to specific features in specific engines or to the overall execution–time robustness issues which affect most full–featured, non–deterministic finite automata–based engines. In a recent article, Cox provides a detailed discussion of the exponential execution–time issues across multiple well–known implementation [14]. Additionally, these issues are well–defined as the risks of regular expression engines [48] [24] [33] and of exponential algorithms in general [15].

Exponential search algorithms aside, the most prominent example of regular expression security research is

the vulnerability [75] found in PCRE's [35] validation of quantifier values. The vulnerability received substantial attention due to its use in attacking Apple's Safari web browser. Other than this discovery, no regular expression security code auditing or testing appears to be noted in a public forum prior to this research.

Additionally, it is important to note that the publicly disclosed vulnerabilities included in this work were presented, as slides, at IT Defense 2008. This presentation provided a discussion of the vulnerabilities discovered as well as demos of most of the findings [58].

1.2 Paper Structure

This paper continues with a discussion of recent examples of insecure context switches. Section 3 examines the survivability of the surveyed regular expression engines in terms of software defects as well as specific inoculation tactics. Section 4 discusses an additional discussion of general inoculation tactics. Section 5 provides conclusions and information on the availability of the fault testing software developed.

2 Insecure Context Switches

There are a number of quite recent examples which exemplify the increased risk of security-sensitive context switches. In particular, the examples below all involve the exposure of software to hostile environments. Often, the exposure created when migrating software to an unexpected threat environment opens a whole area for attack research which was unavailable in the original contexts. Two recent examples are discussed below.

2.1 Malware analysis

Malware, or malicious software, is subjected to extensive analysis by industry and academic researchers. This research is considered vital to the stability of the Internet [64], but analysis may leave the researchers open to attack. In particular, dynamic, or runtime, analysis of malicious code exposes tools to explicitly hostile code.

The use of virtual machines is a standard technique which has long been considered a reasonable approach for dynamic analysis, if the virtual machine and its monitor are securely implemented [26]. However, many researchers were using standard, consumer-targeted virtual machines for this analysis work. Using virtual machines written with an emphasis on consumer software compatibility in this context exposes the software to new, unexpected threats. A full discussion of these threats and some inoculation efforts were presented last year [57]. This analysis led to increased security attention from consumer-grade virtual machine authors. It also raised the general awareness of the risks involved with using virtual machines in unexpected contexts [38] [27].

Even though most dynamic malware analysis techniques involve the use of some form of sandbox, native or virtual machine, systrace, or other approach, there are still many instances where it is considered safe to analyze software without those additional risk-mitigating systems in place. One example involves the analysis of malicious javascript, presented last year [55]. The technique discussed uses command–line javascript interpreters, like NJS [8] and Rhino [54], to evaluate potentially malicious javascript. In theory, this approach is similar to sandboxing the code because its execution context is now not within a live browser. However, this change of execution context is also a change of security context for the command-line javascript interpreters. The change provides a new surface for the attacker to explore and a surface that was not designed with the analysis of explicitly malicious software in mind. For example, NJS may suffer from unreported, or undisclosed [91], vulnerabilities that have little impact in their current context. In the other case, using Rhino may allow the attacker to instantiate and use any number of native Java classes.

2.2 Databases

Databases and their use have evolved in a large number of ways since their inception [56]. Given the importance of databases to Web services, database software has adapted with usage, even if at slightly delayed pace. A major example of this switch of contexts was the move of databases from use in internal, controlled environments to exposure via the Internet. Bina, et. al. discussed some of the first database-wrapping Web interfaces in 1994 [9], but research into effective database use in that context was still considered an active research topic in 1999 [17]. Even with the security concerns taken in to account in Bina's 1994 work, the details of the exposure via a frontend wrapper had not yet been considered. Since then, the backend databases have changed and attacks like SQL injection have become quite prominent. As database-driven Web content became more common, yet another unforseen security context was added. Companies began to offer databases as a service. This new context meant that multiple unrelated users may have access to a database containing each other's information. It also means that the hosting provider must now worry about direct, malicious attacks against the database interface itself [4].

The fundamental relationship of databases to modern software drives its use in to new and unexplored territories. The most recent shift comes with the development of HTML 5, standard browser-based database support [36]. This functionality entails browser-side

databases that are manipulated via server-supplied code. In fact, these are already deployed and can be found in Firefox 3 [28], Adobe Air [2], and Google Gears [32]. As with past context switches, this change opens databases to new threats. The prior risk of trusted users attacking the database now broadens to include the risk of untrusted users. This is a very different environment. Thus far, efforts in this direction have focused on a shared database implementation, SQLite [37]. SQLite has received some security attention from one of the authors of this paper [21] as well as context-aware work from its developers. Even with these steps towards inoculation, this context switch still requires close, continued scrutiny to avoid repeating the events of similar, historic switches.

3 Regular Expression Survivability

Much like databases, regular expressions have a long history in computer science [94] and are heavily intertwined with its future. The ubiquitious reliance on regular expression engines conjoined with the considerable age of many of the implementations provides an ideal case study in survivability. In order to determine the overall survivability of regular expressions, multiple candidates were selected across platform and programming language. Empirical analysis of survivability was performed against each of the candidates using both manual and automated security testing techniques. Results were publicly confirmed. In addition, specific conclusions regarding the inoculation of regular expressions engines were put into practice with, as of writing, success.

Table 1: Regular expression engines analyzed

engine	language	platform
Adobe Flash	Actionscript	multiple
Adobe Reader	Javascript	multiple
GNU grep	shell	Unix–based
ICU	C/C++	multiple
jscript	Javascript	windows
mono	C#	multiple
OCaml	OCaml	multiple
oniguruma	ruby	multiple
PCRE	C	multiple
PostgreSQL	SQL/c	multiple
Python	python	multiple
Spidermonkey	Javascript	multiple
Tcl	Tcl	multiple
Sun's util.regex	Java	multiple

3.0.1 regfuzz

A majority of the testing was performed through the automated generation and evaluation of complex and, potentially, malformed regular expressions. Regular expression patterns were generated iteratively using a seeded, pseudo–random number generator to determine each subsequent term and the overall length of the pattern. The generator started with an innermost term and expanded outward. Both standard and non-standard constructs were supported for analysis. Nearly every implementation supports submatch extraction, character classes, and repeating terms, but some engines, like that of Perl, also support Unicode character classes and back–references. The analysis of engine–specific features directly contributed to our final findings.

The automated testing library is called *regfuzz*. It is written in C along with small test harnesses for each language, where possible. Also, SWIG [74] was used to enable this single library to test additional languages like Perl, python, OCaml, and Tcl. For languages that could not be used with SWIG, such as Javascript and Actionscript, regfuzz was translated manually to the target language. To ensure optimal performance when testing, a pure SQL version of regfuzz was written for use with PostgreSQL, but the performance was an order of magnitude worse than a C–based module.

3.1 Results

All of the regular expression engines tested suffered from the well documented exponential–time execution, or compilation, vulnerabilities [33] [14] [15]. Those findings will not be discussed in detail nor will any undisclosed vulnerabilities. Instead, attacks which affect the overall survivability of regular expressions in a highly connected, or highly exposed, context are discussed. Due to space constraints, detailed discussion is limited to PCRE and Adobe Reader with a note regarding additional findings.

The following classes of vulnerabilities were encountered during this research:

- Stack overflows due to unbounded recursion
- Memory exhaustion attacks resulting in failure or crash
- Unexpected or poorly decoded values, such as back–references
- Invalid expressions, such as impossible submatch conditions
- Compiled byte code, memory–use estimation errors
- Long or infinite compile–time and execution–time attacks

The vulnerabilities of primary concern are those of memory-use estimation errors and poorly decoded values. In those cases, the result is almost always exploitable. Often those vulnerabilities are expressed through improperly handled unicode, escape sequences, and quantifiers.

3.1.1 PCRE

PCRE is a popular BSD licensed regular expression library written in C. It supports a wide range of regular expression-related functionality and has bindings for most popular programming languages. The comprehensive nature of PCRE's feature set only serves to increase the overall potential for bugs.

Numerous exploitable vulnerabilities in PCRE were discovered. Four heap overflow vulnerabilities [76] [77] [80] [82] were found due to the mismanaged escape sequences, incorrect compiled byte code size estimates, and poorly parsed Unicode characters. Two of the vulnerabilities were denial of service attacks [79] [81] due to other incorrect Unicode character and character class handling. Finally, an information leakage vulnerability [78] occurred with certain input bytes were used in non–UTF8 mode.

As PCRE migrated into connected environments accepting remotely–supplied regular expression patterns as input, its security context changed immensely. All of the vulnerabilities discussed below would be mere software defects in its original context. Now, all of these defects are exploitable vulnerabilities. The impact is amplified by its widespread use. PCRE is used and accessible to hostile users in Konqueror, WebKit and Apple Safari [6], Adobe Flash [1], and all browsers where Flash is supported as a plugin. This wide deployment of PCRE into highly connected environments shows both a lack of focus on survivability and the high level of risk that stems from this inattention. The vulnerabilities above were prime candidates for widespread exploitation much like Mark Dowd's recent vulnerability.

3.1.2 Adobe Reader

Adobe Reader is a well–known PDF viewer that is supported on most popular operating systems. In addition, it is often used as a Web browser plugin which is started automatically when a PDF file is visited. It supports regular expressions via the embedded javascript engine. When javascript support is enabled, the default state, all loaded PDFs have access to its facilities. Reader used an old version of Netscape's publicly released javascript engine which predates the more modern Spidermonkey. This dated implementation suffered from a number of vulnerabilities in regular expression handling, in part due to the regular expression code itself, but also due to legacy string handling memory vulnerabilities present in the Netscape javascript engine. The combination of these problems resulted in multiple, potentially exploitable Reader vulnerabilities [3].

Like Adobe Flash, these vulnerabilities may be used to attack Web browsers on all popular operating systems. In addition, malicious PDF files may also be emailed to targets or shared on public file sharing nodes on internal networks with similar effect. The outdated libraries used in Adobe Reader showed very little awareness of the risk associated, and the use of Adobe Reader as a browser plugin undermined the survivability of the browser on the whole. Since this research, Adobe has taken strides to modernize, such as the replacement of its javascript engine, in order to avoid additional attacks against legacy code in this highly connected context.

3.2 Additional findings

A number of other libraries suffered from vulnerable software defects. This includes systems ranging from programming languages to databases. Normally, attacks against programming languages are of little interest since they only impact the software authors. As features, like regular expressions, become more accepted, they are exposed regularly to end-users. This context switch translates software defects, like those below, in to security vulnerabilities.

- Perl [93] suffered from an exploitable heap overflow [87]
- Tcl [59] and PostgreSQL [62] were vulnerable to one out of bounds read [83] and two denial of service attacks [86] [88]
- ICU4C [40] had one heap buffer overflow [85], and an out of bounds memory access [84]
- Boost::Regex++ suffered from two denial of service conditions [90] [89], one of which may have further implications

These findings affect a large number of web application and end-user software, as well as backend services, all due to the exposure of regular expressions to potentially hostile users without deploying any risk mitigating tactics.

3.3 Inoculation

Safe and robust regular expression engine implementation is quite difficult. The expected features of any modern regular expression engine range from submatch extraction and back references to Unicode character

classes. Its complexity and ubiquity are reasons why it is such a compelling attack target.

For completeness, specific inoculation tactics for users of regular expression engines are discussed below. Many of these these techniques are supported in the surveyed engines. Where possible, the authors of this paper aided the adoption. While no quantified benefit analysis is discussed, many of these tactics mitigate the specific risks of the attack classes presented earlier.

Do not expose regular expressions to a hostile environment. This is the ideal scenario from a security perspective, but it is, of course impractical. Regular expressions are a core feature of Javascript which is a core feature of modern web browsers. However, this is not the case in all instances. For example, javascript may be disabled by the user in Adobe Reader rendering the discussed attack vector nonexistent.

Do not compile with -DNDEBUG. In the engines surveyed, many sanity tests in the code bases were done with *assert* calls. These assertions are removed on compilation if the *NDEBUG* value is defined. By not defining this, the software will unexpectedly abort on assertions. This is equivalent to allowing a denial of service attack, but it minimizes the exposure to more severe attacks.

Time and memory bound all phases. The compilation phase and execution phase of regular expression handling should be limited by total time and by total memory. This practice will avoid both exponential execution time and asymmetric memory usage. When used, appropriate timeout behavior must be well-defined. If it is not, the most likely result will be vulnerability to denial of service attacks.

Limit recursion and/or backtracking. Engines, like PCRE, support limiting the maximum amount of recursion for a particular regular expression. This is crucial for avoiding stack overflows when recursive evaluation techniques are used. This is often the case in backtracking algorithms. As with the time and memory bounding, an limitations must be accompanied by appropriate handling to avoid introducing additional issues.

Limit number of NFA states. When regular expressions are compiled to non–deterministic finite automata, limiting the maximum number of states provides an upper bound on both the execution time and memory usage of a particular pattern. This has been used in parser–generators [52] for years, but was only recently added to Mozilla Spidermonkey through the *javascript.options.relimit* option. As part of this research, Tcl and PostgreSQL added state–limiting as a compile–time option to their regular expression engine.

Only use white–listed patterns. The use of white-listed patterns allows the developer to ensure that the engine will behave in a pre-defined manner. In most cases, it also guards against vulnerabilities introduced through future pattern-language features. As part of this research, the authors of GnuPG [46] were contacted and chose to make this exact change. Since GnuPG implements the OpenPGP standard, it allows a trust signature packet which relies on regular expressions for trust signature validation. The range of required regular expression patterns was quite small, but the existing implementation allowed for any possible pattern to be supplied, and even via a remote key server or e-mailed key file.

When authoring patterns, avoid exponential evaluation. When using known good regular expressions, it is important to avoid patterns which may have an exponential evaluation time on specially crafted text input. Friedl [29] supplies examples of expensive expressions, as does Cox [14].

4 General Inoculation

Inoculation techniques are often very specific to the software being inoculated. Specifically, the layers close to the primary software functionality will be the most tailored to its expected contexts. This is seen in the mitigation tactics recommended for regular expression engines. However, this does not preclude more generalized approaches. Below, two well-known techniques are mentioned as well as commentary resulting from the authors' experience with the three context switches discussed earlier. All of these recommendations may aid the the survivability of consumer systems.

Use system facilities: Operating systems, system libraries, and compilers all provide functionality that increase the difficulty of successful, reliable exploitation. These are easy for developers to use and have minimal impact on performance or usability. There is no reason for software to not employ them.

Software fault isolaton: There are numerous techniques for isolating software in sandboxes, from systrace to ptrace [97] and more [98] [99]. Fault isolation is a proven intrusion prevention technique and, in some ways, approximates the original, functionally–isolated context of the software. It allows for centralized software access policy management and a single, smaller source code base for security auditing. In addition, this is a technique that can be deployed by both tech–savvy end–users with coarse–grain control and developers with fine–grain control. In particular, developer integration of sandboxes in software, especially software with plugin support offers huge gains in survivability. This was discussed in the work by Grier, et. al. on secure web browsing [34]. By placing all plugins in a sandboxed environment, SELinux in their case, the potential for successful intrusion was decreased dramatically. This form of integration with existing codebase is extremely desirable.

Context-aware development: Developers must be

aware of the context in which their code will be deployed. With this awareness, they can apply inoculation tactics that will minimize their software's use outside of its expected context. For example, several of the mitigation tactics for regular expressions involve supporting configurable limits on compilation and execution. These limits allow the risks of any future, unknown, contexts to be controlled by the users of the software. In addition, there are a number of excellent references for writing secure software with context-appropriate recommendations [66] [39] [67] [16].

SQLite provides an excellent example for active context-aware development. Instead of solely mitigating risks in legacy code, its transition requires continued attentiveness to the new threat environment. As is, the change required a substantial number of additional checks to the SQL command parsing layer in order to perform robustly.

Context-aware usage: Developers must be aware of the intended security contexts of the software they choose to (re)use. This awareness will enable them to gauge the newly introduced risks and take mitigating steps, such as configuring maximum run–time or sandboxing riskier components.

5 Conclusions

Regular expression engine use is a strong example of the lack of survivability engineering practices in modern Internet-connected software. Legacy software libraries are used without regard for their original context or the risks that are introduced through the high level of connectivity. The disregard for context is only magnified by the lack of any risk mitigation techniques. This behavior undermines the survivability of the entire software system, and in many cases, the entire computing platform.

With the addition of new security contexts, new and dangerous attack vectors will continue to be introduced. These attacks will be barely mitigated by the slow, reactive patching of affected software [7]. Only through the proactive application of inoculation techniques, like software sandboxing, operating system-provided defenses, and security context aware development, will the risks be truly minimized for the end user. Until that point, complex, legacy code used in new contexts will continue to be a highly effective attack vector.

6 Future Work

This work provides a light survey of both past and coming examples of software in new threat environments. An in-depth analysis of the attacks these systems are now subject to and the success or failure of specific inoculation techniques are areas for further research.

In addition, the inoculation of widely used, legacy software in unforseen environments should be continued. In parallel, inoculation tactics should be introduced to software systems in wide-use, such as adding software sandboxing for extensions in browsers. Without this sort of work continuing, the overall survivability of consumer systems will continue to be poor.

7 Acknowledgments

Thanks to multiple anonymous reviewers for their extensive feedback and Google and the Google Security Team for supporting this work. In addition, the authors would like to thank all of the maintainers of regular expression engines whose lives we made difficult with our bug reports.

References

[1] Adobe Inc. APSB07-20. http://www.adobe.com/support/security/bulletins/apsb07-20.html, December 2007.

[2] Adobe Inc. Adobe Air. http://www.adobe.com/products/air/, 2008.

[3] Adobe Inc. APSB08-13. http://www.adobe.com/support/security/bulletins/apsb08-13.html, May 2008.

[4] Anonymous. Mysql maxdb webtool remote stack overflow. http://labs.idefense.com/intelligence/vulnerabilities/display.php?id=234, April 2005.

[5] Apparmor linux application security. http://www.novell.com/linux/security/apparmor/overview.html.

[6] Apple Inc. About the security content of Safari 3.1. http://support.apple.com/kb/HT1315, April 2008.

[7] W. A. Arbaugh, W. L. Fithen, and J. McHugh. Windows of vulnerability: A case study analysis. *Computer*, 33(12):52–59, 2000.

[8] B. Bassett. Njs. http://www.njs-javascript.org/, 2008.

[9] E. J. Bina, R. M. McCool, V. E. Jones, and M. Winslett. Secure access to data over the internet. In *PDIS*, pages 99–102, 1994.

[10] B. Blakley. The emperor's old armor. In *NSPW '96: Proceedings of the 1996 workshop on New security paradigms*, pages 2–16, New York, NY, USA, 1996. ACM.

[11] Y. Chen, R. Dios, A. Mili, L. Wu, and K. Wang. An empirical study of programming language trends. *IEEE Software*, 22(3):72–78, 2005.

[12] S. M. Christey. Vulnerability Type Distribution in CVE. http://attrition.org/pipermail/vim/2006-September/001032.html, September 2006.

[13] M. J. Cox. Vulnerability and threat mitigation features in rhel and fedora. http://www.awe.com/mark/blog/200801070918.html, January 2008.

[14] R. Cox. Regular Expression Matching Can Be Simple And Fast (but is slow in Java, Perl, PHP, Python, Ruby, ...). http://swtch.com/ rsc/regexp/regexp1.html, January 2007.

[15] S. A. Crosby and D. S. Wallach. Denial of service via algorithmic complexity attacks.

[16] N. Daswani, C. Kern, and A. Kesavan. *Foundations of Security: What Every Programmer Needs to Know*. Apress, Berkeley, CA, USA, 2007.

[17] H. Davulcu, J. Freire, M. Kifer, and I. V. Ramakrishnan. A layered architecture for querying dynamic web content. *SIGMOD Rec.*, 28(2):491–502, 1999.

[18] L. P. Deutsch and B. W. Lampson. An online editor. *Commun. ACM*, 10(12):793–799, 1967.

[19] S. Di Paolo and G. Fedon. Subverting ajax. In *23rd Chaos Communication Congress*, December 2006.

[20] M. Dowd. Application-specific attacks: Leveraging the actionscript virtual machine. http://documents.iss.net/whitepapers/IBM_X-Force_WP_final.pdf, 2008.

[21] W. Drewry and D. McNamee. sqlite ¡ 3.4.0 Possible issues. http://bugs.gentoo.org/show_bug.cgi?id=192094, September 2007.

[22] Ellison, Fisher, Linger, Lipson, Longstaff, and Mead. Survivable systems: An emerging discipline. In *Proceedings of the 11th Canadian Information Techology Security Symposium (CITSS)*. Communications Security Establishment, May 1999.

[23] R. J. Ellison, D. A. Fisher, R. C. Linger, H. F. Lipson, T. A. Longstaff, and N. R. Mead. Survivability: Protecting your critical systems. *IEEE Internet Computing*, 03(6):55–63, 1999.

[24] K. Ellul, B. Krawetz, J. Shallit, and M. wei Wang. Regular expressions: new results and open problems. *J. Autom. Lang. Comb.*, 9(2-3):233–256, 2004.

[25] H. Etoh and K. Yoda. Propolice: Improved stack-smashing attack detect on. In *IPSJ SIGNotes Computer SECurity 014(025)*, October 2001.

[26] D. Farmer and W. Venema. *Forensic Discovery*. Addison Wesley Professional, 2004.

[27] S. Farrell. Security boundaries. *IEEE Internet Computing*, 12(1):93–96, 2008.

[28] Firefox Team. Mozilla Firefox 3.0rc2 Release Notes. http://www.mozilla.com/en-US/ firefox/3.0rc2/releasenotes, 2008.

[29] J. E. F. Friedl. *Mastering Regular Expressions*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2002.

[30] C. Gacek. Exploiting domain architectures in software reuse. *SIGSOFT Softw. Eng. Notes*, 20(SI):229–232, 1995.

[31] A. K. Ghosh and J. M. Voas. Inoculating software for survivability. *Commun. ACM*, 42(7):38–44, 1999.

[32] Google Inc. Google Gears. http://gears.google.com/, May 2007.

[33] grep(1). In *GNU Project Manual Pages*, January 2002.

[34] C. Grier, S. Tang, and S. T. King. Secure web browsing with the op web browser. *sp*, 0:402–416, 2008.

[35] P. Hazel. Pcre: Perl-compatible regular expressions. http://www.pcre.org, February 2008.

[36] I. Hickson and D. Hyatt. Html 5: W3c working draft. http://www.w3.org/TR/2008/WD-html5-20080122/, January 2008.

[37] D. R. Hipp and D. Kennedy. sqlite. http://www.sqlite.org.

[38] T. Holz. Security of virtual machines. http://honeyblog.org/archives/109-Security-of-virtual-machines.html, 2007.

[39] M. Howard and D. E. Leblanc. *Writing Secure Code*. Microsoft Press, Redmond, WA, USA, 2002.

[40] International components for unicode. http://www.icu-project.org.

[41] J. Jelinek. -fpie/-fpie/-pie gcc support. http://gcc.gnu.org/ml/gcc-patches/2003-06/msg00140.html.

[42] K. Jiwnani and M. Zelkowitz. Maintaining software with a security perspective, 2002.

[43] E. Jonsson, L. Strömberg, and S. Lindskog. On the functional relation between security and dependability impairments. In *NSPW '99: Proceedings of the 1999 workshop on New security paradigms*, pages 104–111, New York, NY, USA, 2000. ACM.

[44] S. C. Kleene. Representation of events in nerve nets and finite automata. Technical Report RM-704, RAND Corporation, December 1951.

[45] J. C. Knight, R. W. Lubinsky, J. McHugh, and K. J. Sullivan. Architectural approaches to information survivability. Technical report, Charlottesville, VA, USA, 1997.

[46] W. Koch. GnuPG: GNU Privacy Guard. http://www.gnupg.com.

[47] C. W. Krueger. Software reuse. *ACM Comput. Surv.*, 24(2):131–183, 1992.

[48] V. Laurikari. NFAs with Tagged Transitions, Their Conversion to Deterministic Automata and Application to Regular Expressions. In *SPIRE*, pages 181–187, 2000.

[49] B. Lewis, D. LaLiberte, and the GNU Manual Group. *GNU Emacs Lisp Reference Manual*. Free Software Foundation, 675 Mass Ave, Cambridge, MA 02139, 1.03 edition, Dec. 1990.

[50] H. Lipson, N. Mead, and A. Moore. Can we ever build survivable systems from cots components, 2001.

[51] P. Loscocco and S. Smalley. Integrating flexible support for security policies into the linux operating system. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, pages 29–42, Berkeley, CA, USA, 2001. USENIX Association.

[52] T. Mason and D. Brown. *Lex & yacc*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1990.

[53] N. R. Mead, R. C. Linger, J. McHugh, and H. F. Lipson. Managing software development for survivable systems. *Ann. Softw. Eng.*, 11(1):45–78, 2001.

[54] Mozilla Foundation. Rhino: Javascript for java. http://www.mozilla.org/rhino, 2008.

[55] J. Nazario. Reverse engineering malicious javascript. In *CanSecWest 2007*, 2007.

[56] M. L. Neufeld and M. Cornog. Database history: from dinosaurs to compact discs. *J. Am. Soc. Inf. Sci.*, 37(4):183–190, 1986.

[57] T. Ormandy. An empirical study into the security exposure to host of hostile virtualized environments. In *CanSecWest 2007*, 2007.

[58] T. Ormandy and W. Drewry. Regular Exceptions. In *IT Defense 2008*, January 2008.

[59] J. K. Ousterhout. *Tcl and the Tk Toolkit*. Addison Wesley, 1994.

[60] OWASP Team. Owasp download. http://www.owasp.org/index.php/ Category:OWASP_Download.

[61] PaX Team. Pax. http://pax.grsecurity.net.

[62] PostgreSQL. Postgresql. http://www.postgresql.org, 2008.

[63] N. Provos. Improving host security with system call policies. In *SSYM'03: Proceedings of the 12th conference on USENIX Security Symposium*, pages 18–18, Berkeley, CA, USA, 2003. USENIX Association.

[64] N. Provos, D. McNamee, P. Mavrommatis, K. Wang, and N. Modadugu. The ghost in the browser analysis of web-based malware. In *HotBots'07: Proceedings of the first conference on First Workshop on Hot Topics in Understanding Botnets*, pages 4–4, Berkeley, CA, USA, 2007. USENIX Association.

[65] T. Ptacek. Gunnar peterson's os security features chart. http://www.matasano.com/log/611/gunar-petersons-os-security-features-chart/, November 2006.

[66] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. In *Proceedings of the IEEE*, volume 63, pages 1278–1308, 1975.

[67] R. Seacord. Secure coding in c and c++: Of strings and integers. *IEEE Security and Privacy*, 4(1):74–76, 2006.

[68] Secunia. 2006 report. http://secunia.com/gfx/Secunia_Year-end_Report_2006.pdf, 2007.

[69] Secunia. 2007 report. http://secunia.com/gfx/SECUNIA_2007_Report.pdf, 2008.

[70] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *CCS '04: Proceedings of the 11th ACM conference on Computer and communications security*, pages 298–307, New York, NY, USA, 2004. ACM.

[71] K. Siau and T. A. Halpin, editors. *Unified Modeling Language: Systems Analysis, Design and Development Issues*. Idea Group, 2001.

[72] H. Spencer. Re: Regular expression implementations. http://arglist.com/regex/usenet-spencer-1993-08-07.txt, August 1993.

[73] H. Spencer. regex: Henry spencer's regular expression libraries. http://arglist.com/regex/, October 2002.

[74] S. Stanton et al. Simplified wrapper and interface generator. http://www.swig.org, February 2008.

[75] The MITRE Corporation. CAN-2005-2491. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CAN-2005-2491, August 2005.

[76] The MITRE Corporation. CVE-2007-1659. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2007-1659, January 2007.

[77] The MITRE Corporation. CVE-2007-1660. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2007-1660, January 2007.

[78] The MITRE Corporation. CVE-2007-1661. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2007-1661, January 2007.

[79] The MITRE Corporation. CVE-2007-1662. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2007-1662, January 2007.

[80] The MITRE Corporation. CVE-2007-4766. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2007-4766, January 2007.

[81] The MITRE Corporation. CVE-2007-4767. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2007-4767, January 2007.

[82] The MITRE Corporation. CVE-2007-4768. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2007-4768, January 2007.

[83] The MITRE Corporation. CVE-2007-4769. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2007-4769, January 2007.

[84] The MITRE Corporation. CVE-2007-4770. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2007-4770, January 2007.

[85] The MITRE Corporation. CVE-2007-4771. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2007-4771, January 2007.

[86] The MITRE Corporation. CVE-2007-4772. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2007-4772, January 2007.

[87] The MITRE Corporation. CVE-2007-5116. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2007-5116, January 2007.

[88] The MITRE Corporation. CVE-2007-6067. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2007-6067, January 2007.

[89] The MITRE Corporation. CVE-2008-0171. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-0171, January 2007.

[90] The MITRE Corporation. CVE-2008-0172. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-0172, January 2007.

[91] The oCERT Team. oCERT-2008-0010. http://www.ocert.org/advisories/ocert-2008-010.html, July 2008.

[92] The Perl Foundation. perl1. http://dev.perl.org/perl1/dist/, January 1988.

[93] The Perl Foundation. perl. http://www.perl.org/, 2008.

[94] K. Thompson. Programming techniques: Regular expression search algorithm. *Commun. ACM*, 11(6):419–422, 1968.

[95] A. van de Ven. New security enhancments in red hat enterprise linux v.3, update 3. http://www.redhat.com/f/pdf/rhel/WHP0006US_Execshield.pdf.

[96] A. van Hoff. The case for java as a programming language. *IEEE Internet Computing*, 01(1):51–56, 1997.

[97] van 't Noordended, Balogh, Hofman, Brazier, and Tanenbaum. The case for java as a programming language. *IEEE Internet Computing*, 2002.

[98] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. *SIGOPS Oper. Syst. Rev.*, 27(5):203–216, 1993.

[99] R. West and J. Gloudon. User-level sandboxing: a safe and efficient mechanism for extensibility, 2003.

[100] J. Wilander and M. Kamkar. A comparison of publicly available tools for dynamic buffer overflow prevention. In *Proceedings of the 10th Network and Distributed System Security Symposium*, pages 149–162, San Diego, California, February 2003.