ON MITIGATION OF MODERN CYBERCRIME THREATS

Miloš Jovanović[1],
Nikola Rančić[1],
David Davidović[2],
Dragan Mitić[3]

[1]Singidunum University,
Department for postgraduate studies
32 Danijelova Street, Belgrade, Serbia,
[2] Union University,
Faculty of Computer Science,
Knez Mihajlova 6/VI, Belgrade, Serbia
[3]Metropolitan University
Faculty of Information Technology,
Tadeuša Košćuška 63, Belgrade, Serbia

Abstract:
As the infrastructure that humans heavily rely upon grows is dependent on modern technology and the Internet, the damage that can be done by exploiting vulnerabilities in these systems becomes more significant and worrisome. The extent of these threats' possible impact cannot be overstated, as the amount of sensitive information stored in information systems and the actions that they are permitted to perform have been continuously heightening since the beginning of the information age. We present a review of representative examples of security incidents that had put a large number of such systems at risk of abuse, with many of them having withstood documented exploitation "in the wild". We analyze the circumstances that lead to the presence of these security threats, as well as the way they were handled in terms of disclosure and urgent fixes to the affected software. Finally, we also suggest methods which could have possibly prevented these vulnerabilities or lowered their attack surface if they had been applied timely.

Key words:
cybercrime, information security, vulnerability mitigation.

1. BACKGROUND

As Christ (2002) points out, even almost 15 years ago, the growth of the Internet (more precisely, the metric considered is the total count of reachable web servers on publicly accessible IP ranges), exponential in nature, presented unique challenges to the technology underlying it. Today, the communication standards and protocols used at the time are long superseded by their newer versions or solutions that are completely redesigned from the ground up, which is only natural for quickly evolving technology that needs to keep pace with the explosion of its use. The same holds for software used as the backbone of these communications: from network drivers that provide advanced routing features to web and e-mail servers operating at the top of the OSI networking model (ISO/IEC 7498-1:1994). Some notable examples include:

- Protocols and standards:
  - HTTP/1.0, the first iteration of the Hypertext Transfer Protocol, has been abandoned in favor of HTTP/1.1 which is now the dominant method of serving web pages on the Internet. HTTP/1.1 is, in turn, being built upon and improved with the introduction of efforts such as Google's SPDY (Chromium

Correspondence:
Miloš Jovanović

e-mail:
mjovanovic@openlink.rs

DOI: 10.15308/Sinteza-2016-137-142

Project 2009) and the recently standardized HTTP/2 (Belshe *et al.*, 2015) which is, as of the time of writing, supported by 6.7% of the top 10 million websites by popularity (W3Techs 2016a).

- Plain TCP as an unencrypted stream transport protocol for WAN communication is largely being abandoned in favor of protocols providing authenticity and confidentiality of the transmitted communication. These protocols themselves have gone through a large number of iterations, many of them because of identified security flaws. These include, among others, SSLv3 (Freier *et al.* 2011) and TLS, whose current standardized version is TLSv1.2 (Dierks and Rescorla 2008).
- HTML, CSS and JavaScript, the standards used for development of website front-ends, have undergone possibly the most significant changes of all, mainly because of the constant demand for more immersive and advanced web experiences by consumers, companies and enthusiasts alike. Among these, the most recent iterations are HTML5 (W3C 2014), CSS3 and ECMAScript 6 (ECMA 2015). It is important to note that for the former two, the version number is more of a formal nature, as new features are constantly being added and refined by means of feature proposals, browser adoption, and, finally, standardization.

◆ Software:

- Networking drivers and routing strategies used by operating systems running on servers and on embedded routers, firewalls and other networking-capable devices have undergone heavy improvement in order to adapt to higher throughput and responsiveness requirements without the need for heavily increasing the needed resources.
- The landscape of HTTP server market share, once unanimously led by Apache (which still leads with over 50% of total usage), is now significantly more fragmented, with newer technology (most notably the Nginx web server) taking up more than 30% of the total amount (W3Techs 2016b).
- E-mail servers, CMS solutions, server monitoring, analysis and deployment tools, and many others used today bear no resemblance to the technology comparably used in the past.

It is thus evident that the requirement of fast-paced evolution and iterative improvements, also reflected in recent trends within software engineering itself, such as agile software development (Cohen *et al.* 2003) exists and dictates much of the development within this field. However, such a steep rate of innovation leaves a lot of space for mistakes and insufficient quality assurance of these types of products.

Wall (2007) suggests, and the authors agree, that the types of crime brought about by the information age present a more persistent and worrying threat than is perceived by many entities, most significantly, the organizations and individuals that develop and maintain software and standards needed to keep the rate of technological advancement constant or growing, and at the same time keep that software reasonably secure and immune to a wide range of security exploits.

Nevertheless, it is clear that the industry has mechanisms in place to appropriately deal with the existence and mitigation of such threats, but the level of adoption and enforcement of those mechanisms still has room for improvement, all in the interest of minimizing risk for consumers and organizations relying on these software systems for personal, business, financial or other needs.

2. VULNERABILITIES IN THE RECENT YEARS

As the topic of cybercrime is more popular than ever before among technology journalists, hobbyists, privacy advocates, security researchers, and other groups, it is of no surprise that high-profile security vulnerabilities garner much attention in the public. Below is given a brief and, unfortunately, incomplete list of some more heavily publicized and threatening instances of such oversights.

CVE-2014-0160 (Heartbleed)

A discussion touching on these issues cannot be complete without first mentioning one of the most significant and publicly known software security issues in the recent history of computing: CVE-2014-0160, or more commonly (and memorably) known as *Heartbleed* or the *Heartbleed bug*.

CVE-2014-0160 is a bug in OpenSSL, a software library aiming to provide a complete solution for implementing SSL/TLS protocols mentioned earlier in this article, both on the server and client side. OpenSSL, at the highest level, supports transparent secure communica-

tion between endpoints on a network where an adversary might be able to passively capture or actively alter the traffic (for a more in-depth description of the security guarantees these protocols make, refer to the relevant RFCs by the IETF). To achieve this goal, OpenSSL has had to provide many other features: it includes a library implementing a vast array of cryptographic primitives (ranging from symmetric block cipher and elliptic curve cryptography implementations to secure key exchange protocols) and support for parsing, verification and manipulation of X.509 certificates, among others. Bearing this in mind, it is not surprising that OpenSSL is a big library with many separate organizational units interacting in complex ways.

The root cause was a failure of code that dealt with packets regarding the TLS *heartbeat* extension to explicitly check if the advertised size of a byte string in a packet matched its real size. This effectively lead to an out-of-bounds memory read and subsequent disclosure of this information to the attacker, meaning that software using vulnerable OpenSSL versions would leak contents of arbitrary memory locations, leading to possibly disastrous scenarios (MITRE 2014a). The OpenSSL project was informed of the issue beforehand and had supplied a fix for the bug before it was disclosed to the public.

As a vast majority of web servers use OpenSSL as their library of choice for implementing HTTPS (HTTP through SSL/TLS) support, the impact of this bug was extremely high (Durumeric *et al.*, 2014) so high, in fact, that it spawned forks of OpenSSL such as LibreSSL (by OpenBSD developers) and BoringSSL (by Google) that aim to trim down OpenSSL's codebase and employ other methods of reducing the risk of such issues in the future.

CVE-2014-6271 (Shellshock or the Bash bug)

A vulnerability comparable to *Heartbleed* in scope and impact is surely CVE-2014-6271, which has also had considerable media exposure and thus gained the, among the public perhaps more recognizable, nickname of *Shellshock* or the *Bash bug*.

GNU bash (*Bourne-again shell*) is a Unix shell and language first released in 1989, as a replacement to the then-dominant but non-free *Bourne shell.* A shell has been an ubiquitous component of almost all Unix-based systems from the beginnings of Unix - it is used as the basic text-based user-system interface, a script language and a "surrogate process" capable of spawning new processes in a precisely defined environment. The latter use plays a key role on the discussed vulnerability, as there are very few nontrivial programs that do not spawn the default shell at some point, and GNU bash, being the default shell on a lot of systems, presents an attractive and widely critical attack surface.

The bug is caused by a flaw in the parsing logic of GNU bash, whereby specially crafted and non-sanitized environment variables could cause arbitrary code execution in the context of the user and process spawning the shell: namely, the feature of function definition within environment variables could be abused to execute code regardless of whether or not the function is actually called (MITRE, 2014c). The GNU project was responsibly informed and had supplied a fix before the knowledge of the bug was made public.

Some web servers (those using CGI, a dated technology for dynamically generated web content that is still being used), e-mail clients and similar software did rely on the default shell for part of their functionality and supplied it with user-provided input, which led to a direct remote code execution vulnerability. The count of publicly accessible and servers vulnerable to this issue was considerable (Delamore and Ko, 2015).

Other examples

As a thorough analysis of other vulnerabilities is beyond the scope of this article, some examples which were not as publicized and whose impact was not as high, but nonetheless appropriately illustrate the nature of contemporary security-critical software bugs, will be given in more compact form.

- CVE-2016-0800, dubbed the *DROWN attack* by its creators, uses an SSLv2 and TLS enabled server to perform a cross-protocol attack that retrieves the plaintext of passively collected communication between the server and a user. It achieves this by using the SSLv2 endpoint as a Bleichenbacher padding oracle in order to unmask session keys negotiated with the TLS endpoint and a victim, and also relies on a previously known bug in OpenSSL to speed up the attack. According to the authors, more than 20% of tested hosts were vulnerable at the time of disclosure (Aviram *et al.*, 2016; MITRE, 2016).
- CVE-2015-0235, also known as *GHOST*, is a buffer overflow bug in *glibc*, the most widely used implementation of the C standard library, which almost every computer program directly or indirectly relies upon on modern systems. The bug was located inside a function whose task was to

perform network address lookups, and as such exposed almost all software that used the network in some way to risk. Qualys, the company that discovered the bug, claims that they have created a proof-of-concept exploit that uses this vulnerability to remotely execute arbitrary code via *Exim*, a popular e-mail server (MITRE, 2015). It is worth noting that the exploit in question was never made public.

- CVE-2014-3153, a serious vulnerability interesting for both its impact and difference from the ones previously mentioned, is a privilege escalation exploit within the Linux kernel where a non-privileged attacker could gain privileged access to a system, abusing a bug in the *futex* (*fast user-space mutex*) subsystem of the kernel where a paused privileged worker thread could be made to, by manipulating its stack, transfer execution to arbitrary locations when woken up (MITRE, 2014b).

3. MITIGATION

When looking at these vulnerabilities—subjectively—in hindsight, it would appear reasonable to believe that they are caused by carelessness on part of the developers. The steps to exploit most of them are not complex and are well within the budget or knowledge of any competent computer programmer or security researcher. However, these kinds of mistakes do happen in some quantity regardless of the amount of care or expertise of the development team or individual. Thus, it also seems reasonable for programmers to expect that there are the ways to of early discovery (before vulnerable code is released) and that tools and workflows they use should do their best in order to promote safe coding practices and prevent or minimize the impact of such issues.

In that spirit, we have identified some key areas which can possibly lead to minimization of these types of threats. Worthy of note is that it is exactly these kinds of threats that can be and are used, as has been demonstrated by Wall (2007), for nefarious purposes and serious compromise of individual privacy and safety and putting the whole businesses in jeopardy. Additionally, they pose a threat to state-level security and enable further and even more serious criminal activities.

Software testing

Automated unit and integration testing of software products has been practiced and its importance has been known for a long amount of time (Zhu *et al.*, 1997). The development of advanced fuzzy testing utilities such as the *American Fuzzy Lop* (Zalewski, 2016) has made it easier to find unexpected bugs and issues not covered by traditional unit and integration testing methodologies, and tools for static analysis of programs have come a long way to predict and pinpoint possible causes of bugs early in the development process of a particular feature or component. However, these kinds of tools are alleged not to be used by developers as often as they ought to (Johnson *et al.*, 2013).

On the other hand, 100% code/branch coverage requires a lot of effort in order to be maintained at that level, and programmers view writing tests as a notoriously unimaginative use of their time. This issue is something that is better dealt with economically, by giving programmers working on security-critical code better incentives for maintaining a high level of code coverage, or by (optimally) delegating that matter to competent quality assurance engineers whose only focus would then be to keep the product thoroughly automatically tested after every change.

Security audits

Independent researchers, mostly for economic-, reputation- or enthusiasm-driven reasons, often conduct security audits of popular software that is heavily relied upon for security, or is a component in a considerable amount of systems where it can cause further security issues if vulnerable to attacks in some way. Often, these reviews are performed by companies and organizations specializing in software security, cybercrime prevention and related fields, in order to gain recognition for uncovering one or more vulnerabilities and possible attacks, or for purely ideological reasons of improving the state-of-the-art.

In the wake of "Snowden revelations", cybersecurity has become a politically polarized subject, a characterization that can possibly be put to good purpose: security-minded individuals and organizations alongside those who care about the cause of secure computing and mitigation of related threats can sponsor (through donations, crowdfunding and similar means) big audits and reviews of existing and relied upon software in order to achieve further guarantees of its safety and proper design.

Of course, this is not limited to full-scale audits of software. Code review by a maintainer or a more experienced developer should be mandatory, not optional, and should be enforced on every code commit. This

allows developers who are more innately familiar with the codebase to spot early problems that may arise due to complex interactions between separate modules, which cannot be achieved with an incomplete understanding of the project's code.

Safer programming languages

OpenSSL, a library that has been the target of many recently uncovered vulnerabilities, is written in C, and so is the entirety of almost all Unix-compatible kernels currently in active use, which are the cornerstone of a majority of web servers in the world at 68% (W3Techs, 2016c), smartphone devices, routers, hardware firewalls and even home appliances.

C is a language first designed in 1978 and not receiving any significant, fundamental standards update since the present day. It is perhaps obvious that the kind of programming language which, for instance, does not provide any memory safety guarantees and which is, in comparison to presently available languages, a thin abstraction around low-level assembly, is ill-suited for the kinds of uses it is being put to today. With the availability of many safer, significantly more modern, less error-prone and mostly just as performant programming languages, a long-term goal of using them instead of lower-level alternatives (where possible) can lower the number of critical security vulnerabilities that are now being discovered on a monthly basis.

Among languages that are better suited for this particular use-case, the authors would like to highlight *Rust* (Mozilla, 2016), an open-source effort by *Mozilla*, which is explicitly designed for safe network and systems programming, and by design prevents several classes of behavior that are known to have been the biggest sources of critical vulnerabilities to date (namely, memory safety violations and race conditions).

Economical incentives

Software that is used to power most of the modern web is, in big part, free/libre software or open-source (among many examples are Linux, Apache, MySQL, PHP, Redis, *etc.*), which tremendously helps their users by allowing them to modify and tweak their behavior, benefit from the work of the community as a whole and remove the financial barriers to using fast, robust and safe software. On the other hand, developers working on free/libre and open-source software are mostly not compensated for their work, and consider it a hobby, yet are able to create functional and well-performing tools that are used and relied upon by big organizations and causes.

Donating and encouraging donations to these projects, especially by entities that have significantly benefited from their use, can help in the long-term by creating a more favorable position and allowing the maintainers and active members of the project to dedicate more time to further refining the piece of software, testing it and improving its quality in terms of both performance and safety.

4. CONCLUSION

As we have seen, unfortunately, there is no shortage of high-profile security vulnerabilities that open up attack surfaces for serious compromise of information systems used nowadays. Most of these vulnerabilities have been brought about by either lack of manpower on complex software projects (as is mostly the case with OpenSSL), sparsely tested legacy code (as is mostly the case with Bash), and other factors and combinations thereof.

We believe that the proposed high-level means of mitigating these kinds of threats can prove fruitful in the long run, as the principles behind them have been well known and thoroughly proven effective in the software engineering and security industry for a long time, though their application, in our opinion, has not been widespread enough to prevent these issues or significantly lower their impact.

Finally, we believe that, while our proposed methods involve more up-front investment, the economic and humanitarian damage effected by the multitude of security-critical bugs that compromise individual, as well as corporate privacy and integrity, far outweighs the possible downsides of implementing these security practices. As such, we encourage organizations, individuals, software developers, project maintainers and other involved parties to consider the facts presented and draw their own conclusions about the usefulness of our proposed approach.

REFERENCES

Aviram, N., Schinzel, S., Somorovsky, J., Heninger, N., Dankel, M., Steube, J., Valenta, L., Adrian, D., Halderman, J. A., Dukhovni, V., Käsper, E., Cohney, S., Engels, S., Paar, C. & Shavitt, Y. (2016). *DROWN: Breaking TLS using SSLv2.* Retreived March 14, 2016. From https://drownattack.com/drown-attack-paper.pdf.

Belshe, M., Peon, R. & Thomson, E. M. (2015). *Hypertext Transfer Protocol Version 2 (HTTP/2)*, RFC 7540.

Christ, H. D. K. M. (2002). *Lay Internet Usage: An Empirical Study with Implications for Electronic Commerce and Public Policy*.

Chromium Project (2009). *SPDY: An experimental protocol for a faster web*. Retrieved March 14, 2016. From https://dev.chromium.org/spdy/spdy-whitepaper.

Cohen, D., Lindvall, M., & Costa, P. (2003). *Agile software development*. DACS SOAR Report, 11.

Delamore, B., & Ko, R. K. (2015). *A Global, Empirical Analysis of the Shellshock Vulnerability in Web Applications*. In Trustcom/BigDataSE/ISPA, 2015 IEEE, 1, 1129-1135.

Dierks, T. & Rescorla, E. (2008). *The Transport Layer Security (TLS) Protocol Version 1.2*, RFC 5246.

Durumeric, Z., Kasten, J., Adrian, D., Halderman, J. A., Bailey, M., Li, F. & Paxson, V. (2014, November). *The matter of Heartbleed*. In Proceedings of the 2014 Conference on Internet Measurement Conference (pp. 475-488). ACM.

ECMA (2015). *ECMA-262 6th Edition: ECMAScript® 2015 Language Specification*. Retrieved March 14, 2016. From http://www.ecma-international.org/ecma-262/6.0/.

Freier, A., Karlton, P., & Kocher, P. (2011). *The Secure Sockets Layer (SSL) Protocol Version 3.0*, RFC 6101.

ISO/IEC (1994). ISO/IEC 7498-1:1994. *Information technology – Open systems interconnection – Basic reference model: The basic model*.

Johnson, B., Song, Y., Murphy-Hill, E., & Bowdidge, R. (2013, May). *Why don't software developers use static analysis tools to find bugs?*. In Software Engineering (ICSE), 2013 35th International Conference on (pp. 672-681). IEEE.

MITRE (2014a). *CVE-2014-0160*. Retrieved March 14, 2016. From https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2014-0160.

MITRE (2014b). *CVE-2014-3153*. Retrieved March 14, 2016. From https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2014-3153.

MITRE (2014c). *CVE-2014-6271*. Retrieved March 14, 2016. From https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2014-6271.

MITRE (2015). *CVE-2015-0235*. Retrieved March 14, 2016. From https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2015-0235.

MITRE (2016). *CVE-2016-0800*. Retrieved March 14, 2016. From https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2016-0800.

Mozilla (2016). *The Rust Programming Language*. Retrieved March 14, 2016. From https://www.rust-lang.org/.

W3C (2014). *HTML5: A vocabulary and associated APIs for HTML and XHTML*. Retrieved March 14, 2016. From https://www.w3.org/TR/2014/REC-html5-20141028/.

W3Techs (2016a). *Usage of HTTP/2 for websites*. Retrieved March 14, 2016. From http://w3techs.com/technologies/details/ce-http2/all/all.

W3Techs (2016b). *Usage of web servers for websites*. Retrieved March 14, 2016. From http://w3techs.com/technologies/overview/web_server/all.

W3Techs (2016c). *Usage of operating systems for websites*. Retrieved March 14, 2016. From http://w3techs.com/technologies/overview/operating_system/all.

Wall, D. (2007). *Cybercrime: The transformation of crime in the information age*.

Zalewski, M. (2016). *American Fuzzy Lop*. Retrieved March 14, 2016. From http://lcamtuf.coredump.cx/afl/.

Zhu, H., Hall, P. A., & May, J. H. (1997). *Software unit test coverage and adequacy*. ACM Computing Surveys (csur), 29(4), 366-427.