Was the 2006 Debian SSL Debacle a System Accident?

George Markowsky
University of Maine, Orono ME 0469-5711, USA, markov@maine.edu, DrGM.us

Abstract – In this paper we examine in detail the Debian OpenSSL Debacle from the perspective of a system accident, a concept derived from the work of Charles Perrow [1]. This event left users of Debian and its derivatives with seriously compromised cryptographic capabilities. We identify some common failings that might be problematic in other software development projects and offer some suggestions to help develop code more securely.

Keywords – Debian; system accident; SSL; cryptography; security breach; software engineering

I. INTRODUCTION

In most cases, when something goes wrong, we look for someone to blame. The underlying assumption is that someone must have been the direct cause of each particular mishap. In [1], Charles Perrow introduced the concept of a system accident, something he also called a normal accident. The basic idea is that while a serious accident might include a number of smaller events the serious accident occurred because the system allowed the smaller events to interact in a manner that strengthened their total contribution. While his analysis targets physical systems, his emphasis on looking at the various components of a system is valuable for virtual systems as well. In this paper, we develop the concept of a system accident and apply it in the analysis of a little known incident that occurred in 2006 in which the Debian SSL library was compromised for two years before someone noticed that it had been compromised. We conclude with some suggestions on improving the system that led to this system accident.

II. THE DEBIAN SSL DEBACLE

The discussion of the Debian SSL Debacle in this paper is based on papers by Ahmad [2], blog entries by Cox [3] and Schneier [4], the video by Bello and Bertacchini [5], and the video by Applebaum, Zovi and Nohl [6].

Ironically, the Debian SSL Debacle was caused by Debian developers trying to do a good thing. As is well known, memory errors are a fertile breeding ground for software failures of all types and often lead to breaches in cybersecurity. The Open Source community has developed tools to help find memory errors in software. In particular, “Valgrind” [7] is one such tool.

One of the developers running Valgrind on the Debian source code received a message indicating the use of an uninitialized variable in the function MD_Update in two places. Normally, this is an error in programming, but in this case the uninitialized variable was used together with other components to increase the amount of randomness in the OpenSSL module of the Debian operating system. In one location in the code, the use of MD_Update brought in other critically important sources of randomness in addition to the randomness imported from the uninitialized variable. Removing this instance of the MD_Update function critically damaged the OpenSSL module.

The OpenSSL module is used to encrypt communication for the operating system. For example, a person signing into a “secure” website is typically dependent on the OpenSSL module for providing good cryptographic strength. In general, the more randomness in a cryptographic system the better, so reducing the amount of randomness is a serious step to take.

The developer discussed with other Debian developers and also corresponded with OpenSSL developers. We shall examine the conversation in more detail in Section IV. He received a message approving the removal of the lines in question for debugging purposes, but he interpreted it as an approval to remove the lines in general, so he removed them from the program by commenting them out.

The main consequence of this action was to essentially set the stage so that at most 32,768 passwords were possible. While this seems like many passwords to the average person, this is a very small number for a computer to try in a “brute-force attack.” The lines were commented out in 2006, and it was not until 2008 that the error was discovered and the weakness of the resulting cryptographic system was established. Thus for about two years the cryptographic capabilities of Debian Linux were severely compromised. Hence the title of Ahmad’s paper “Two Years of Broken Crypto” [2].

III. SYSTEM ACCIDENTS

Perrow’s book [1] contains detailed accounts of many industrial accidents as well as accidents in various other areas. Among the most fascinating accounts are those
involve ships colliding in the ocean. A number of these collisions start off with the ships on courses that would not lead to a collision. Unfortunately, actions taken by the crew cause the collision. An example of such a collision is shown in Figure 1 which comes from [1, p. 210]. Clearly, a collision is something both crews are eager to avoid, yet their actions led to a collision. Clearly, there is something about the marine transportation system that contributed to the collisions. There may, indeed, be failings on the part of the crew, but there are deeper factors at work that led to the accident.

Perrow uses the term system accident or normal accident throughout his book [1] but nowhere does he give a clear and concise definition. Furthermore, his focus is on physical systems and some of the factors he focuses on are less relevant to virtual systems. We will review the appropriate terms from [1] and then propose how to interpret them for virtual systems.

First, Perrow spends a bit of time distinguishing between “incidents” and “accidents.” Both of these “involve damage to a defined system that disrupts the ongoing or future output of that system” [1, p. 64]. Furthermore, Perrow divides systems into four levels: part, unit, subsystem and system. An incident is an event that occurs at the first two levels and an accident is an event that occurs at the last two levels. This leads Perrow to the follow formal definition [1, p. 66]. Note that the acronym ESF stands for engineered safety feature.

We are ready for a formal definition. An accident is a failure in a subsystem, or the system as a whole, that damages more than one unit and in doing so disrupts the ongoing or future output of the system. An incident involves damage that is limited to parts or a unit, whether the failure disrupts the system or not. By disrupt we mean the output ceases or decreases to the extent that prompt repairs will be required. Since we have drawn a dividing line between the unit and the subsystem, and since many of the ESFs are clustered around that dividing line, it will often mean that an ESF will be one of the components that fails.

There are many features of the preceding definition that do not apply to the Debian OpenSSL Debacle. For example, the error that was introduced never interfered with the operation of the Debian operating system. In fact, the operating system ran without incident for two years before anyone noticed the error. In addition, it is not clear who was damaged by this bug and to what extent. We will address this issue further in Section V. Clearly, the potential was there for massive cybersecurity breaches but we have no easy way to tell which security breaches were the result of this failure. In the case of physical systems and accidents like factory explosions and ship collisions it is often obvious what is damaged and to what extent. Of course, software systems can also cease to function as a result of errors or attacks, but it is not unusual for large software systems to have many errors and to function competently. Of course, many of the errors are often minor and occur in very limited circumstances.

One of the major points that Perrow makes in [1] is that people, in particular system owners, like to find a scapegoat so that they are not obligated to fix the system. Often this is because system owners believe that it is cheaper to deal with the occasional mishap rather than fix the system [1, p. 67]. Perrow also has some very interesting points to make about the complexity of systems and the tight coupling of systems. There is no doubt that many software systems such as operating systems are extremely complex. Furthermore, there are few complex physical systems that do not depend on a complex software system for control.

In 2008 Perrow wrote an unpublished paper [8] dealing with software failures. While his analysis of technical issues leaves much to be desired, this paper is a fascinating collection of software failures of varying severity and is worth reading. The distinction between incident and accident is a valuable one even for virtual systems. We will use the term incident to describe a failure that has caused or is likely to be more of an inconvenience or nuisance than a serious failure. We will reserve the term
accident for a failure that has caused or has the potential to cause serious damage or loss.

Perrow believes that serious accidents are often not just the result of one person’s mistakes. Rather, they are often the result of a sequence of minor mistakes which combine to produce the serious accident. In this, they are aided by features of the relevant system that make the accident more likely. For example, in [1] Perrow shows how the operator is often blamed even though various safety devices did not work properly or gave misleading information. He spends lot of time in his analysis of the Three Mile Island nuclear accident [1, Chap. 1] showing how the operators were unable to get correct information from some of the gauges, yet they were blamed entirely for the accident.

We want to make the concept of a booby trap central to our definition of a system accident. A booby trap is designed to hurt someone who gets in its way. Booby traps can be set deliberately or unintentionally. Booby traps are often set by property owners to protect their property. Many polities have laws or regulations against the setting of booby traps because the final result of a booby trap is unpredictable and is much worse than the consequence of the crime the booby trap is designed to prevent. Many people realize that booby traps such as wiring shotguns to doors are not intelligent things to construct. In fact, it is not unusual for property owners to be killed by their own booby traps [9].

For software systems we define a booby trap as some feature that makes it more likely that a user will make an error. One famous example of a software booby trap is the famous loss of the Mars Climate Orbiter spacecraft in 1999 [10]. The problem was that the two engineering teams that worked on the software for this system used different systems of measurements. In particular, one team used the metric system and the other team used the imperial system for measurements. As one would expect, at some point someone forgot to make the proper conversion and as a result a $125 million satellite was lost. It is clear that having two teams working with two different measurement systems is a booby trap. While one can try to blame “operator error” for the accident, it was clearly a system accident because the system was set up to make such an accident extremely likely. Despite this costly error, NASA continues to use both measurement systems for projects, although in 2007 it made the commitment to use the metric system for all operations on the lunar surface [11].

We propose the following definition: a software failure will be considered a system accident if it has serious consequences or the potential of having serious consequences and was caused by one or more booby traps. Our primary purpose is to identify booby traps in the software creation system, because even if they lead to minor errors in one circumstance, they can lead to more serious consequences in other circumstances.

IV. THE DEBIAN SSL DEBACLE REVISITED

In this section we will revisit the Debian OpenSSL Debacle from the point of view of finding booby traps in the open source development system that might be present in other systems as well. For convenient reference we will use DOD to refer to the Debian OpenSSL Debacle.

The first point to consider is the common lack of proper commenting in code. Many programmers either do not know how, don’t have the time, or are too lazy to write proper comments that really explain their reasoning and what the code does. They also tend not to point out booby traps in their code. This problem affects all types of software both proprietary and open source. This is Booby Trap 1. This booby trap was sprung in the DOD as illustrated by the fact that the developers who were fixing the “bug” did not properly understand the nature of the code they were working on.

Closely related to Booby Trap 1, is the writing of overly clever code. This is code which does something in a very efficient and elegant manner and often combines multiple operations into one. Such code needs a lot of commenting and is often hard to understand for other programmers. It is often hard to understand for the programmer who wrote it once the creative insight passes. We call this Booby Trap 2 and it was sprung in the DOD as discussed below.

While being proactive and finding errors in software before they manifest themselves is a good thing, there is a fundamental danger in using automated tools because these tools do not understand the logic of programs and perform their analysis at a very low level. Such tools can report errors, which are not truly errors. This is Booby Trap 3, and it was sprung in the DOD as seen from in the correspondence between Debian developers [12].

Valgrind has the property that once it finds an “error” it will track the effects of that error as they propagate through the entire software system. On the one hand this seems like a good idea, but in practice it leads to a large number of error messages. Experience shows that providing humans with too many warnings tends to make them ignore the warnings or even shut the system down. Perrow [1] describes many instances where safety systems were shut down before accidents because they were overwhelming the operators. This is Booby Trap 4.

It was sprung in the DOD as illustrated in [12].

There is another booby trap associated with the use of powerful software testing tools. This is the potential to lead developers to venture into parts of the code that they do not understand well. This is Booby Trap 5. It was sprung in the DOD as illustrated in [12]. In this correspondence one of the developers notes that two lines in particular are the cause of the Valgrind error messages. He also notes that the function of these lines is to add “uninitialized numbers to the pool to create random numbers.” Unfortunately, this developer did not
What I currently see as best option is to actually comment out those 2 lines of code. But I have no idea what effect this really has on the RNG. The only effect I see is that the pool might receive less entropy. But on the other hand, I'm not even sure how much entropy some uninitialised [sic] data has.

What do you people think about removing those 2 lines of code?

Figure 2. Portion of a Letter to the OpenSSL Project

Not much. If it helps with debugging, I'm in favor of removing them. (However the last time I checked, valgrind reported thousands of bogus error messages. Has that situation gotten better?)

Figure 3. A Reply to the Letter from Figure 2

fully understand that other sources of randomness were involved as well. The rest of the discussion in [12] turned into a technical discussion about using Valgrind, and the key point about contributing entropy was buried in the other discussion. One of the developers in the discussion was not completely comfortable with the discussion and realized that he was out of his depth in accessing the entropy issue and he contacted the OpenSSL developers [13]. In his note he made the statements shown in Figure 2.

There is another key point that needs to be made. The Debian developers identified two lines of code involved in generating the multitude of Valgrind errors. Both lines introduced an indeterminate amount of entropy from the uninitialized memory location, but one of the lines also introduced entropy from other sources such as system time, the PID, the UID and the random number generator. By focusing on the uninitialized variable, the developers overlooked all the other sources of randomness. Had these sources of entropy been introduced in separate lines, only the entropy coming from the uninitialized memory location would have been eliminated from the program. This is how Booby Trap 2 was sprung. The developers were led into this trap by inadequate commenting of the code which was too clever.

The first reply to the letter in Figure [13] is reproduced in Figure 3. It includes just one paragraph from the original letter which is emphasized. There are several problems with the reply. First, the reply begins with the phrase “Not much.” It is not clear what this refers to. If it refers to the sentence just above, then it suggests that the uninitialized variables do not add much entropy to the random number generator (RNG). This would support commenting out the lines in question. On the other hand, the original correspondence concluded with the line “What do you people think about removing those 2 lines of code?” It is possible that the writer was responding to that question and was stating that he did not think much of removing the two lines of code. Another ambiguity in the reply is the statement “If it helps with debugging, I’m in favor of removing them.” Did the author mean that he was in favor of removing the lines but only for debugging purposes, or was he supporting the removal of the lines period? In any event, the Debian developer decided that this was the approval that he needed to remove the lines of code. The ambiguity in this communication was Booby Trap 6.

Figure 4. The Rate of Updating Vulnerable Certificates

V. DISCOVERY, MITIGATION AND CONSEQUENCES

Debian released Debian Security Advisory DSA-1571 [15] in 2008. It stated that Lucianon Bello discovered the flaw in the OpenSSL package used by Debian. It suggested some remediation such as regenerating various security keys. The announcement should have been enhanced to impress upon people the importance of this error. A variety of exploits aimed at this bug can be found in [5], [6], [16], [17], and [18]. The sources just cited point to sources that should be consulted for additional exploits. DSA-1571 failed to address the true scope of the problem. Compromises affected not only Debian systems with the faulty software, but other systems that engaged in certain types of interactions with compromised computers. Reference [19] contains a discussion of weak keys, but this discussion would be very difficult for a non-expert to follow. The failure to completely explain all consequences of this vulnerability and the failure to more widely alert the user community is Booby Trap 7.

There was another booby trap set by the Debian organization in the way that they handled the announcement
of the vulnerability. In particular, they posted the vulnerability patch on May 7, 2008 but withheld the public announcement of the vulnerability until May 13, 2008. This is Booby Trap 8. The problem here is that there are skilled people who read code changes and who would understand the significance of this error even without the announcement. Not seeing the announcement at the same time as the patch, they would realize that there would be a window of opportunity to brute-force attack systems. Reference [6] reports that there was a sharp increase in the number of brute-force attacks against many hosts during the period between May 7 and May 13.

The inappropriate commenting out of two lines reduced the key space to a maximum of $2^{15} = 32,768$ keys. The total number of keys is greater because the set of 32,768 depends on the system used. In any event, generating the total number of keys for all systems is feasible with limited equipment. [6] even demonstrates how to use Amazon Cloud Services to generate the keys in a relative hurry for under $25.

References [2], [3], [4], [5], [6], and [18] all discuss the consequences of this failure. Of particular interest is the graph in Figure 4 which comes from [18]. It appears that after nearly six months more than 40% of the vulnerable certificates had not been updated. This is Booby Trap 9.

VI. CONCLUSIONS AND SUGGESTIONS FOR IMPROVEMENT

In this paper we identified nine booby traps that led to the DOD and contributed to worsening the consequences. We list them below with a brief description and some suggestions for dealing with each booby trap. It is clear that these booby traps are not unique to the DOD and perhaps the lessons learned here can be helpful elsewhere.

1) Poor Commenting. This is a standard problem in coding and it is not clear that we are making progress. Figure 5 shows a more recent version of the code that was modified to create the DOD. Note that while there is a comment warning people not to remove a particular instance of MD_Update, there is no explanation of why an uninitialized variable is being used. Furthermore, there are several other references to MD_Update in the same code section but there is no explanation of what these other calls are achieving.

2) Overly Clever Coding. Programmers should realize that while very clever coding might save some space and coding time, it is very difficult to understand
and is a fertile breeding ground of errors. Figure 5 shows there are now more calls to MD_Update so the functionality has been separated to some extent, but no information is given about what each call is doing.

3) Uncritical Use of Automated Software Analysis Tools. Figure 6 which comes from [5] shows the comparison of code before and after the modification. The right half of the figure has the changes in green. The presentation is a bit deceptive because the effect of the introduced lines is to remove the lines dealing with MD_Update from the program. We need to make it more obvious to an observer that the changes have affected the lines containing MD_Update.

4) Overwhelming Error Messages. Thought needs to be given on how to demonstrate all weaknesses in some code without overwhelming the person using the automated tool.

5) Repairs by Nonexperts. If the Federal government wants to help improve US cybersecurity it should consider offering code reviews for critical software. Clearly, it is not reasonable for the Federal government to review all code, but Debian and its derivatives such as Ubuntu are very popular and are the basis of a significant number of servers on the Internet.

6) Ambiguous Communication. Perhaps a more formal process can be instituted here to make sure that all questions are posed and answered unambiguously. It appears that the standard method of just mailing to a site and having people selectively reply to sections of the e-mail is fraught with danger.

7) Overtechnical Announcements with Poor Circulation. We need to get more people with public relations and communications skills active in the open source community. These people must be made to feel welcome and not put down by the technical community since they have an important job to perform.

8) Posting Patches Prematurely. Clearly, it is recommended that organizations not publicly post patches before they announce vulnerabilities. Perhaps they can post patches only to vetted customers to give them a chance to update their systems.

9) User Community Not Taking Cybersecurity Seriously Enough or Perhaps not Having the Resources to Deal With Critical Issues. This is a challenging problem that deserves a separate discussion.

REFERENCES

[9] Ben Feinstein, “Loaded Dice: SSH Key Exchange & the Debian OpenSSL PRNG Vulnerability,” DEFCON 16, August 8-10, 2008, Las Vegas, NV, http://www.youtube.com/watch?v=yXr7KBC3G3I. The presentation is a bit deceptive because the effect of the introduced lines is to remove the MD_Update lines containing MD_Update.

Figure 6. Diff Obscuring High Level Understanding