Why Good Software Engineering Practices Often Do Not Produce Secure Software

Carlisle Adams, Guy-Vincent Jourdan

School of Information Technology and Engineering (SITE)
University of Ottawa
800 King Edward Avenue
Ottawa, Ontario, Canada, K1N 6N5
{cadams , gvj}@site.uottawa.ca

Abstract

This paper discusses some of the reasons why software engineering practices are insufficient for producing software that is suitable for use in a security-sensitive environment. It suggests the need for education among system designers/developers, and the use of a team of software security specialists in all software development projects.

1. Introduction

Over the past three decades, a significant amount of research has been done in the field of software engineering, leading to a much better understanding of how software projects should be conducted. Yet, software security seems to have at best marginally improved during that same period. Much of the research related specifically to security and software has focused on implementation of cryptographic algorithms, secure protocols, and software protection mechanisms (including access control, antivirus tools, and the like). In these areas, much progress has been achieved. However, the security-relevant aspects of software applications themselves (i.e., as executing programs) has received less attention from the academic community. A major portion of the active research published in this area has come from outside academia.

Many books about software engineering available today are essentially silent about software security. We believe that this should not be the case. The field of software engineering must address the security implications of creating software if there is ever to be a hope of producing and developing less-vulnerable software. Software built following the best software engineering practices available today has virtually no chance to be "secure software"; instead, it is highly likely to exhibit several more or less critical security flaws, making that software unfit for use in a security sensitive environment.

In this paper, we look at some of the reasons why software engineering practices do not typically help to create secure software. We first show that classical testing is ill-suited for catching security related bugs. We then look at two other well-known software security issues: injection and session management. In both cases, we show that software engineering practices have little to offer in terms of protection against these common security problems.

The paper concludes with some directions that would help software engineering practices to produce secure software. Until such practices are in place, it is unreasonable to expect much in terms of security from simply following good security engineering methodology.

2. Passing the tests with flying colors

If there is one type of security related issue that should be prevented when sound software engineering practices are followed, it is software security issues due to bugs. After all, bug-related security issues are just that: plain old bugs in the code. A good test plan aimed at eradicating most bugs in the system, therefore, should readily take care of the security bugs.

In fact, it doesn't. Test plans will generally not find security related bugs, and even if these bugs are found, they will most likely remain uncorrected.

2.1. Examples of bug related security issues

Simple bugs in the application may have very severe security consequences. Consider two very common problems: denial of service and buffer overflow.

2.1.1. Denial of service. Denial of service attacks have received a lot of attention lately [1]. With most modern software having a multi-tier architecture (in particular, virtually all Web-based applications), denial of service is becoming an almost ubiquitous type of attack, leaving no-one safe.

Much of the focus and attention today is directed toward the difficult issue of distributed denial of service (also known as DDoS), but for the sake of illustration, consider the very simple case of a denial of service due to a crash of one of the tiers of the system (i.e., without massive flooding involved so that the attack comes from a single computer, for example after one single malicious request).

If an attacker is able to locate one poorly handled request anywhere in the application, then this attacker can disrupt the service at will and can easily render the whole system essentially non-functional. All it takes is this one unexpected input and the service goes down for seconds, minutes, or more.

Being able to remotely crash a service because of some mishandling in the processing of a request is thus a serious, bug-related security issue.

2.1.2. Buffer overflow. "Exploitable" buffer overflows and related bugs are usually a step up from denial of service bugs. With an exploitable buffer overflow, the attackers can do more than crash the service: there is the possibility of uploading arbitrary code and having this code executed by the process running the attacked software, under the privileges of the original software program. It is not the aim of this paper to explain how this type of attack can be carried out; suffice it to say that it has until recently constituted the bulk of the "critical security advisory" alerts, and is very widespread and well known across the security community. See [2, 3] for an overview.

Here again, this type of issue is entirely the result of a simple bug in the code (in the case described here, typically a simple out-of-bounds array access) and is often a critical security issue (in that running an application having such a bug may give the attacker full access to the machine running the application).

2.2. Usual "proper" testing will not catch these bugs

It is important to note that a normal test plan that follows generally accepted software engineering practices will not catch the bugs that have been outlined above. The reason for this is that normal test plans have a strong bias toward functionality testing and, at best, graceful handling of unintentional ill-conceived user input. Because of the general acknowledgment that complete coverage is not possible, the testing phase does little to protect against input that significantly departs from such cases.

2.2.1. Unexpected situations. A typical denial of service attack consumes some resource until none is left available. Typical testing, on the other hand, considers the expected usage of the resource, and may add some comfortable padding to the test so that the software is tested under some multiple of the expected load. These tests ensure that in the worst "normal" situation, the service will remain available and reasonably responsive.

The denial of service attack does not, however, consider what a bad "normal" situation is, but rather what is the worst possible situation that can intentionally be created. It is, for example, easy to open hundreds of concurrent connections from a single machine (using an average desktop computer over a simple home based internet connection).

No test plan that is not specifically designed to take denial of service attacks into account will base its metrics on the maximum load that can be intentionally put on the service using readily available computing means, rather than some small multiple of the expected load.

Failing to test for the maximum load case, a test plan is not able to determine whether or not the resulting software offers any protection whatsoever against the simplest denial of service attacks.

2.2.2. Unexpected inputs. Another type of security situation can occur as a result of unexpected inputs. A typical good test plan will make sure that the software has some resistance to wrong input (in particular, erroneous, missing, or malformed data entry).

However, an attacker seeking a denial of service or buffer overflow attack will try to feed the software with totally unexpected values at every possible opportunity. Instead of inputting one erroneous character, the attacker will send massively corrupted data, where every single value is unexpected, defined relationships between input fields are ignored, and data not normally supplied by the user is modified if at all possible.

A particularly common attack consists of feeding the software with extra large inputs, typically several

kilobytes where the expected input is a few characters. This type of probing is done using tools that not only make it easy to submit very large amounts of data, but also bypass any user interface limitations and controls imposed by the application.

Again, we find that test plans typically do not account for totally arbitrary input. In fact, testing activities are almost always run with the "normal" user in mind. If the user interface permits the entry of 30 characters in one field, a good test plan will try entering 0 characters, 30 characters, 50 characters, "strange" characters, and maybe even non-alphanumeric characters. But test plans typically do not use a tool to bypass the interface entirely, or to submit more than 10,000 characters to see what happens.

Failing to perform such verifications, the test plan is not able ensure that the resulting software is resistant to the very type of attack that has been most widely publicized over the past 10 years!

2.2.3. "Tunnel vision" testing. In the two simple examples provided above, the problem was not that testing was not done properly. It was as good a test as you will get following widely accepted software engineering practices. The problem was rather that security-related bugs are bugs that are not typically covered by the scope of testing done today. Bug-related security issues arise from very particular types of bugs and software must be tested for these bugs in addition to conventional testing. This will not happen unless software engineers and software testers have security in mind when they design and test their code.

2.2. Even if caught, security-related bugs may go uncorrected

In the previous section, we showed that a security related bug is likely not to be caught using common testing practices. In fact, the situation is worse than this: even if the bug is somehow "stumbled upon" and becomes known to the development/testing team (but the severe security sensitivity of the bug is not understood), chances are that it will remain uncorrected anyway.

Most companies having good software engineering practices will provide for a defect tracking system. When a bug is found, a report is filled out and recorded in the system. The report is then evaluated for clarity and completeness and a decision is then taken to address the issue, ignore the issue, or postpone the decision.

If the project is run according to good software engineering practices, then a bug-related report should never be ignored. Instead, a commitment will be made to resolve the issue in due course, and a priority will be associated with the resolution of the case.

If the bug is a security-related bug of the sort described above, it will not be a bug that prevents normal usage of the software. In fact, according to the committee assessing the report, it will not be associated with any possible real situation at all. Faced with a report stating that "opening several thousand concurrent sessions crashes the system", the committee will note that the projected peak usage is in the tens of concurrent users and assign the bug a very low priority. Faced with a report explaining that "inputting 10,000 characters in a particular field seems to create some problems", the committee will note that (a) the expected maximum input for that field is 30 characters, and (b) the user interface indeed enforces the 30-character limit; the committee will then conclude that this bug is relatively unimportant and, again, assign it a very low priority.

With most busy development projects, low-priority bugs do not get fixed because there are likely to be a large number of defect reports that have been assigned higher priorities, and there are also a number of change requests that are going to be seen as much more important (from the perspective of satisfying customers or staying ahead of competitors, for example).

Thus, even if the bug is discovered, it will probably not be addressed. The security implications of the bug are not known to the development and testing teams and so the low priority assigned to the bug will never be elevated.

3. Command or Code Injection

In the cases described so far, the problem leading to a software security issue was clearly a bug in the code. We have shown why the mechanisms put in place by sound software engineering practices to correct bugs typically do not detect these types of bugs but, at least, such mechanisms do exist. Thus, it can be argued that the scope and/or the techniques used should be expanded, but there is an existing place to catch these problems.

There are, however, many scenarios that are not so clearly addressed by standard software engineering practices. For these types of problems, a simple re-evaluation of current practices and improvement of existing techniques will not suffice. Additional, security-specific steps have to be incorporated.

One such scenario is "injections" [4]. Injection problems have been known for many years. For example, it used to be very common to have "command

injection" vulnerabilities in applications deployed on standard Unix-based systems in the 1980's. Over the years, awareness of the problem has grown, but at the same time many more types of injections have been discovered and exploited.

We first review several "injection" type problems, and then look at why prevention of these problems is not achieved by current software engineering practices.

3.1. Examples of injection scenarios

Injection scenarios can be summarized as follows: the attacked software is used as a "doorway" to get some data entered into the system. The data is harmless to the software, but is harmful to some other components of the underlying system. The injection may be direct (e.g., command injection or SQL injection), indirect (e.g., cross-site scripting), or more complicated (e.g., second-order injection), but the overall attack methodology is the same.

3.1.1. Command injection. As mentioned above, this type of injection has existed for at least 2 decades. A typical scenario is the following: the attacked software gets some data, usually as user input, and part of this data is passed as a parameter to another program. Under Unix, a very typical example would be a system prompting for an email address and then proceeding to send an email by starting a shell and calling the "mail" program, simply passing the received email address to the called program.

The potential security issue is simple: if as part of the email address the attacker has entered some predefined characters and followed the right syntax, then any command can be concatenated to the call to the email program. The shell will thus execute whatever command the attacker the attacker wants it to (the only constraint being the privileges under which the shell was started).

3.1.2. SQL injection. This type of injection has flourished with the development of internet-based applications, which typically have some interaction with a relational database. SQL injections are very similar to command injections: the attacked software uses some user-supplied information to construct an SQL query that is then sent to the database management system [4, 5]. The attacker can therefore insert within the data the necessary special characters and, by following the right syntax, an SQL command chosen by the attacker will be executed by the database.

Here again, when such an injection is possible, the only mitigating factor is the privileges under which the legitimate SQL command was to be executed.

3.1.3. Cross site scripting (XSS). The injections above are direct attacks on the system itself. Injection-based attacks can also be used to indirectly attack other users of the system. The most common such attack is known as "cross site scripting", or XSS [4, 6].

With an XSS attack, the attacker uses the system to inject data that will later be delivered to other users and do some harm there. A typical example would be a bulletin board. The attacker may write a message to the bulletin board, and the message will later be displayed to all users of the bulletin board. This can be transformed into an attack if, again, there is a way to craft a special message that will then be interpreted as a command by the tool displaying the message to a user. Usually, the tool used is a Web browser, and Web browser "commands" are scripts that are included in the Web page. The vulnerability thus lies in the possibility that the attacker may include scripting information as part of the submitted message. If this is possible, then every user viewing the attacker's message will also get that attacker's chosen scripting code executed on his/her machine.

In this type of injection attack, the only migrating factor is any intrinsic limitations imposed by user machines on scripts coming from the host's site. Therefore, because many users do not turn off scripting, in a user population of any reasonable size, there is a high probability that the attack will be successful on at least one user.

3.1.4. Second order injections. Injection type attacks can in fact be more complicated than those outlined above. So far, we have seen how it can be used to perform an attack from within the system itself (server side), and how it can be used to attack the users of the system (client side). In both these cases, the attack is carried out via some tools that are normal, expected components of the whole system (a database management system or a user browser, for example). But injection attacks can also be used to carry out attacks that do not directly use any software related to the system.

An example of a second order injection [7] would be, in a Web environment, to request a page that does not exist, but send script information as part of the requested URL. The system (the Web server in this case) will return a "page not found" error to the attacker. However, it will also likely record the erroneous request in the Web server log. Now, it is

commonplace for the Webmaster of the system to regularly peruse the logs, in order to see trends in usage and detect potential problems (including, ironically, indication of attacks). Due to sheer volume, the Webmaster will not likely look at the raw logs, but will instead use a tool to create a "user friendly" report (for example, in HTML format). This is where the injection opportunity lies: if the carefully crafted URL contains scripting information that is going to be incorporated as part of the report, then by merely opening the report the webmaster will cause the script to be automatically executed.

This type of second-order attack is generally more difficult to carry out and is less controllable. When the attack will occur (if at all) is unknown to the attacker and may be several days away. On the other hand, if the attack succeeds, it has a high probability of being carried out from within the premises of the attacked system and under the privileges of a Webmaster or system administrator. A system susceptible to such a problem should therefore not be run in a security sensitive environment.

3.2. Usual software engineering methods do not prevent injection flaws

All the injection attacks described above work basically according to the same scenario: the attacked software is used as a means to deliver the attack to another part of the system, or to another user. The attack is harmless to the primary software, and causes no damage to it, either immediately or at a later time.

It may be argued that an injection flaw is not a "bug", at least not in the traditional sense. It is certainly not a bug in the component that is eventually attacked. This component is behaving in the normal, expected way. A command shell is, by design, able to execute several commands in a row. Similarly, support for complex SQL statements is not a bug, but rather is a useful feature of any modern database management system. Web browsers may offer the possibility to turn off scripting (and this would thwart the XSS attack as described above), but leaving scripting turned on is not a bug either – indeed, many users will choose to do so in order to gain enhanced functionality from their browsers.

Clearly, if the responsibility is to fall anywhere, it should perhaps be on the shoulders of the software that is used to put the data into the system. However, that data is totally harmless to the software itself. Furthermore, it may be argued that software developers would have a hard time protecting other software and users from attacks, especially when they have little or no control over some of these other components (e.g., the user's browser).

From the viewpoint of software engineering, the difficulty of pinpointing an obvious source to the problem explains in part why, again, normal and sound techniques do not ensure protection against these security flaws. The issue is that the problem is not really located in one single place. It is a system-wide vulnerability, and can even be beyond the system for second-order injections. As the software is being designed and built, there is no clear point in most software engineering methods that provides an opportunity for such a global evaluation of the security implications of very specific, low-level technical details.

We must also point out that if normal testing is not able to detect security-related bugs impacting the very software it is testing (as shown in Section 2), it is even less likely to detect a security-related "bug" that is not really a "bug", and that only impacts other system components.

4. Session management flaws

The last example of this study concerns software security flaws that are due to architectural problems. Of all the problems examined so far, this type seems to be the least likely to be addressed by proper usage of current software engineering techniques.

We illustrate the issue using some of the security problems linked to session management, and we then evaluate why the problems in question will not be addressed by simply following current software engineering methods.

4.1. Examples of session management problems

It is now very common for an application to offer a Web front end. This is a convenient and fast way for developers to provide remote access to the application.

One of the main problems with Web based applications is that the HTTP protocol on which the solution is based is a stateless, session-less protocol, meant to deliver one Web page after another. Most applications, however, do require the notion of state and session. Consequently, various "solutions" (such as the use of client-side stored "cookies") have been developed to simulate a session. However, this is indeed a simulated session: sessions do not really exist.

4.1.1. Session hijacking. The most obvious problem with Web session simulation is session

"hijacking" [4]. Loosely speaking, a session is simulated by generating a unique number and associating that number with the session; every request coming from the "owner" of the session will be sent along with the session number. On the server side, the program will use that number to "recognize" the user and infer the current state and values of server variables.

The important point is that from the server viewpoint, the only way to identify the session is by the session number. If a request comes along with that number, then as far as the server is concerned, the request belongs to that session.

A session hijacking attack consists of sending a request to the server accompanied with another user's session number, and consequently impersonating that user to the server.

The security implication of session hijacking is clear: by definition, from the server viewpoint, the attacker is the user whose session number has been hijacked. The user's data will be freely accessed by the attacker, and conversely any action done by the attacker will be recorded as having been done by the user.

All it takes is for the attacker to "steal" the user's session number, or to guess it, or to randomly choose numbers until an active current session number is found. Needless to say, actual attacks based on all three alternatives have been reported.

Session hijacking is a smaller problem than it once was, not because application developers got better in this particular area, but because existing programming environments and application servers provide a session number generation service that is now reasonably secure.

4.1.2. Session "riding". Much more difficult to address than session hijacking is an attack known as "session riding" [8] (or "Cross-Site Request Forgeries" [9], amongst other names).

In a session riding attack, the attacker doesn't send a request to the server using another user's session number. Rather, the attacker manages to get the user to directly send the request, and therefore "rides" on the current session of the user.

Session numbers are typically sent via cookies. In practice, this means that the session number is automatically sent along with any request to the originating Web site. If the attacker wants to "ride" on a user's session and have this user request a particular URL with particular parameters, all that is needed is to convince the user to somehow "click" on the requested URL. As already mentioned, the session number will be sent along automatically.

Convincing a user to click on the attacker's desired URL might seem difficult. However, the notion of "clicking" is very broad. For example, doing an HTTP GET on an image within a Web page is equivalent to clicking. So all the attacker has to do is to convince the user to view a Web page containing an "image" which is in fact a reference to the URL of the attacker's choice. By viewing this unrelated page, the user will indeed emit a request for the URL, along with his/her session number.

There is still the problem of getting an arbitrary user to visualize a Web page while otherwise having a session currently active on the target web site. This, too, can sometimes be very easily achieved! Assume that the system under attack offers some means of displaying user data (e.g., with some message board functionality, either provided as a side tool, or because this is part of the application). The attacker can then post a message with an embedded "image" pointing at the target URL. Every user of the system viewing the message will subsequently request the target URL, passing along his/her current session number.

As an example, imagine the Web site of a bank. From that site, an authenticated user can transfer money to another account by filling out a form identifying the account number to which the money should be transferred and the amount to be transferred. Submitting the form sends a request to a particular URL. The user is identified from his session number and the money is transferred from that user account to the requested account. Imagine now that this bank's on-line system also has a chat room. All an attacker has to do is open an account, then post a message in the chat room containing a link (for example via a fake image) to the "transfer money" URL, passing his own account number – and some fixed amount of money to be transferred – as parameters. Every bank user logged onto the system that happens to open the page containing the attacker's message will automatically "transfer" the requested amount into the attacker's account!

4.2. Software engineering methods will typically not prevent session management flaws

The problems illustrated in this section are architectural problems with the system. As with Section 3, what has been described here is not a "bug" in the software; on the contrary, the software is doing precisely what it was designed to do. However, the

flaws can have serious security implications. Even if the incidence of session hijacking has been reduced (by the incorporation into most modern tools of a session manager, which frees the implementation team from the headache of a home-grown secure implementation), it is safe to say that the vast majority of Web-based applications today are completely vulnerable to "session riding" attacks, as described above. And, again, it is very likely the case that Web applications developed following the most stringent software engineering techniques available will be no less vulnerable than the rest.

With this attack, we have arguably raised the bar quite a bit. There is no particular application-level bug; there is no dangerous data or scripting program stored on the system; there is no other component to protect. In fact, there is what seems to be a legitimate request, which is indeed coming from the user. In some sense, we have to protect the user from himself/herself, without of course rendering the entire application useless. To take our example, we have to somehow enforce that a user-issued request to transfer money is processed only if the user "truly means it"! Needless to say, as the example clearly demonstrates, failure to catch and prevent this category of problems renders the application unfit for use in a security sensitive environment.

5. Some directions that may help

After such a bleak overview, what are the solutions that may be used when building software that is intended to be run in a security sensitive environment?

5.1. Solutions exist for the problems listed in the paper

The first thing to clarify is that each issue listed in this paper does have clear, well understood, efficient solutions. It is not our goal to list these solutions here, but we are in no way suggesting that these problems are particularly hard to address once they are identified. In fact, there is a variety of tools that can help identify some of the problems listed here, for example through a static analysis of the source code (see, for example, UC Berkeley's BLAST [10] or Microsoft's SLAM [11]). Our point is rather that current software engineering techniques are not equipped to identify these problems in the first place.

5.2. Toward more general solutions

What seems to be the main shortcoming of software engineering methods with respect to secure system development is the lack of recognition of the unique nature of software security related issues.

Many of the problems listed above do not get addressed because software engineering plans simply "don't look there" at all. Even in the one case that is well-addressed (testing for bugs), the special character of security-related bugs is not recognized. Typically, the only way a security-related bug will be identified is when that bug happens to also have an impact in the areas that are typically looked at (such as functionality or usability).

Security must be understood as an important issue, spanning the complete system, and it must be recognized as being broader than cryptography, secure protocols, or data access.

5.2.1. Education. Raising the security awareness of software engineers and software project managers is clearly the number one priority. No professional today should be ignorant of the problems of the type listed above, but should instead have a precise understanding of the source and solution to these problems. No student should graduate with a software engineering degree while completely lacking the minimum background understanding of security issues.

The idea is definitely not to transform every software engineer into a security specialist, but to ensure that every software engineer and every software manager has some technical understanding of the problem and knows that the question must be addressed somehow, as part of the normal software development activity.

5.2.2. Security team. A significant percentage of software programs have a user interface. It is generally understood by the profession that getting the user interface right is a specialized task and, consequently, any professional software organization has a team of user interface specialists. These specialists are part of the software building process. If good software engineering practices are followed, this team is involved early in the software development life cycle, and stays "in the loop" for as long as it is needed.

It is time to reach similar conclusions with respect to the security aspects of any software that is built. A team of software security specialists should be created and should be involved in every project. In fact, it can be argued that no software should today be installed or maintained in a security sensitive environment if that software was not at least scrutinized and tested by such a team of software security specialists.

It is worth noting that the growing emphasis on security over the past few years at Microsoft Corporation has led to similar conclusions, as recently described in [12]. Each project gets assigned a "security advisor" who is the contact point between the project's developers and the security team, and who serves as a security resource and guide thorough the software development life cycle. The security advisor accompanies the development during the complete life cycle, and involves the rest of the security team from time to time, as needed. The security advisor is involved right from the requirements phase in order to maximize the effect and minimize the impact of producing (more) secure software. During the design phase, this expert conducts threat modeling, designs the security architecture, evaluates and minimizes the application attack surface, and evaluates the need to define project-specific security ship criteria. During the implementation phase, the security advisor audits the code, uses static and dynamic security analysis tools on the code being produced, and makes sure that the development team follows secure coding and testing standards. During the verification phase, a thorough security analysis is performed on the near-shippable code (usually involving other members of the security team). Finally, once the release phase is reached, an "independent" security review is conducted by other experts from the security team and an assessment of the overall security level of the application is done. The security advisor will keep following the product during the maintenance phase, since security-related issues are still likely to appear from time to time.

Only once such a team is integrated into the process can we have reasonable hope that proper security oriented testing will be performed as part of the test plan, including testing for the flaws listed here (and other flaws that are also well known), as well as testing for the flaws that are doubtless going to appear in the future.

The user interface team focuses solely on user interface issues and does not interfere in the implementation of the actual functionalities of the system. In a similar way, the software security team will focus solely on software security analysis and provide this global security-oriented evaluation of the system, thus finally providing an opportunity to identify problems such as the ones listed above.

5.3. Long term or short term issue?

There are reasonable grounds to suggest that many of the software security issues that are so widespread today are mainly due to a lack of maturity in the software building process and to a lack of understanding of the area of software security. Indeed, some of the topmost software security issues of the previous decade have diminished in current software development. The most obvious problems are now well known, and technological evolution has also played a role. For example, we have already mentioned the session management support offered by most systems now, and we can of course list the common native support for strong security solutions. In particular, widespread use of a new generation of programming languages has drastically reduced exploitable buffer overflow issues, even though the average software engineer has still no clear understanding of the problem. Similarly, the latest versions of the most popular programming environments provide very effective protection against many of the injection problems listed above.

At the same time, new types of security issues keep appearing and so it seems unlikely that the stream of new problems is going to dry up any time soon.

It is conceivable that at some point in the future the need for a software security specialist team (for every software development organization that can afford it) will vanish. At this moment, however, such a team seems to be the only practical way to have even a small chance of ending up with software that does not have critical security flaws.

6. Discussion

The primary focus of this paper has not been to suggest that the field of software engineering is flawed in any particular way. Rather, it has been to highlight the fact that software engineering does not necessarily produce secure code. Building secure systems (sometimes referred to as "security engineering") overlaps with software engineering in some areas, but the two disciplines are not equivalent.

The focus of software engineering is *functionality*: code should be produced that does what it is supposed to do, with development costs and time that are reasonable and predictable. Because of this focus on functionality, particular consideration is given to the "normal" user of the eventual system, with some attention also paid to the naïve user (so that the system will be tolerant of some mistakes and inadvertent input).

Security engineering, on the other hand, has *dependability* as its focus: code should be produced that starts off, and remains, consistent with an explicitly-specified security policy. This focus on dependability leads to consideration not just of

"normal" and naïve users, but particularly of malicious users (those who apply all their resources and every means available to them to make the system diverge from the security policy in whatever way possible, typically for their own gain).

Deliberately malicious users are not part of the picture in the mind of the typical software engineer, and so it is not surprising that good software engineering practices offer little protection against such users of the system. However, as more and more of our world is controlled (or at least manipulated) by software, the need to take the malicious user into account cannot be overemphasized. It is imperative that security engineering principles take a more prominent role in the design and development of systems, particularly those that deliver critical infrastructure services to society.

7. Conclusions

In this paper, we have demonstrated the need for general software security awareness amongst the developers and managers of software systems. We point out the need for systematic software security assessments covering all aspects of a software system and performed by a team of specialized security experts before any system can be used in a security sensitive environment.

Using specific examples, we have shown that the current safeguard mechanisms in place – good software engineering practices and specialized engineers in charge of particular aspects of security (such as software protection mechanisms, cryptographic algorithms, or secure protocols) – fail to protect complete systems against a wide range of security critical issues.

8. References

[1] Cheswick, W. R., Bellovin, S. M., Rubin, A. D., *Firewalls and Internet Security: Repelling the Wily Hacker*, Addison-Wesley, February 2003 (second edition). ISBN: 0-201-63466-X

[2] E. Levy (Aleph One), "Smashing The Stack For Fun And Profit", *Phrack magazine*, Volume 7, Issue 49, November 1996. Available from http://www.phrack.org

[3] Viega, J., McGraw, G., *Building Secure Software: How to Avoid Security Problems the Right Way*, Addison-Wesley, September 2001. ISBN: 0-201-72152-X

[4] Curphey, M. *et al.*, *OWASP Guide to Building Secure Web Applications*, Open Web Application Security Project (OWASP), September 2002. Available from http://www.owasp.org/documentation/guide/guide_about.html

[5] SPI Dynamics, Inc., *SQL Injection: Are Your Web Applications Vulnerable?*, 2002. Available from http://www.spidynamics.com/papers/SQLInjectionWhitePaper.pdf

[6] Cgisecurity.com, *The Cross Site Scripting FAQ*, May 2002. Available from http://www.cgisecurity.com/articles/xss-faq.txt

[7] Ollmann, G., *Second Order Code Injection Attacks: Advanced Code Injection Techniques and Testing Procedures*, Next Generation Security Software, Ltd., November 2004. Available from http://www.nextgenss.com/papers/SecondOrderCodeInjection.pdf

[8] Schreiber, T., *Session Riding: A Widespread Vulnerability in Today's Web Applications*, SecureNet GmbH, December 2004. Available from http://www.securenet.de/papers/Session_Riding.pdf

[9] Message sent by "Peter W" on the BugTraq mailing list, *Cross-Site Request Forgeries (Re: The Dangers of Allowing Users to Post Images)*, June 15, 2001. Available from http://www.securityfocus.com/archive/1/191390

[10] Berkeley Lazy Abstraction Software Verification Tool (BLAST). Available from http://www-cad.eecs.berkeley.edu/~rupak/blast

[11] The Software, Languages, Analysis and Model checking project (SLAM). Available from http://www.research.microsoft.com/slam

[12] Lipner, S., Howard, M., "The Trustworthy Computing Security Development Lifecycle", Microsoft Corporation, March 2005. Available from http://msdn.microsoft.com/security/sdl