CLASP Resources

The CLASP Resources provide introductions to the most important concepts that underlie the CLASP process. These concepts are referenced from the role-based overviews (see section Role-Based View) and are relied upon throughout the rest of the process. For example, CLASP Resource D discusses the core security services: authorization; authentication; confidentiality; integrity; availability, accountability; non-repudiation.

Even if you have already had exposure to these services, it is recommended to examine the CLASP discussion since these concepts are relied upon heavily, particularly in requirements definition and analysis.

A: Basic Principles in Application Security

This CLASP Resource is meant as a set of basic principles for all members of your application-security project.

Ethics in Secure-Software Development

Software development organizations should behave ethically as a whole, but should not expect that their individual components will.

In so far as security goes, it is ethical not to expose a user to security risks that are known and will not be obvious to the user, without clearly informing the user of those risks (and preferably, mitigation strategies).

It is also ethical to provide users with a specific privacy policy for use of their personal information in a timely manner so that they can act to avoid undesired use of that information, if they so desire. Additionally, if you change a privacy policy, the user should be given the explicit choice either to accept the change or to have his personal data expunged.

Additionally, if you have a system that is compromised on which user data resides, it is ethical to inform users of the breach in privacy. If the data resides in the state of California, this is required by law. Similar regulations may apply in other jurisdictions.

Do not expect that all other people on the development team will be ethical. Insiders play a significant factor in over 50% of corporate security breaches. Particularly at risks are those employees that are silently disgruntled or have recently left the company.

Insider Threats as the Weak Link

Most development organizations overlook “insider” risks — i.e., those users with inside access to the application, whether it be in deployment or development. For example, when planning for deployments it is easy to assume “a firewall will be there,” although, even when true, there are many techniques for circumventing a firewall.

Most development organizations completely ignore the risks from the guy in the next cube or on the next floor, the risks from the secretaries and the janitors, the risks from those who have recently quit or been fired. This, despite yearly numbers from the C*omputer Crime and Security Survey* performed by the Computer Security Institute and the FBI, which shows that over half of all security incidents have an inside angle.

This suggests that trusting the people around you isn’t good enough. Not only might people be disgruntled or susceptible to bribe that you may not expect, but people are often susceptible to accidentally giving insider help by falling victim to *social engineering* attacks.

Social engineering is when an attacker uses his social skills (generally involving deception) to meet his security ends. For example, he may convince technical support that he is a particular user who has forgotten his password, and get the password changed over the phone. This is why many people have moved to systems where passwords can be reset automatically only using a “secret question” — although secret questions are a bit too repetitive. If someone is being targeted, it is often easy to figure out the mother’s maiden name, the person’s favorite color, and the name of his or her pets.

Assume the Network is Compromised

There are many categories of attack that can be launched by attackers with access to any network media that can see application traffic. Many people assume wrongly that such attacks are not feasible, assuming that it is “difficult to get in the middle of network communications,” especially when most communications are from ISP to ISP.

One misconception is that an attacker actually needs to “be in the middle” for a network attack to be successful. Ethernet is a shared medium, and it turns out that attacks can be launched if the bad guy is on one of the shared segments that will see the traffic. Generally, the greatest risk lies in the local networks that the endpoints use.

Many people think that plugging into a network via a switch will prevent against the threat on the local network. Unfortunately, that is not true, as switches can have their traffic intercepted and monitored using a technique called ARP spoofing. And even if this problem were easily addressed, there are always attacks on the physical media that tend to be easy to perform.

As for router infrastructure, remember that most routers run software. For example, Cisco’s routers run IOS, an operating system written in C that has had exploitable conditions found in it in the past. It may occasionally be reasonable for an attacker to truly be “in the middle.”

Another misconception is that network-level attacks are difficult to perform. There are tools that easily automate them. For example, “dsniff” will automate many attacks, including man-in-the-middle eavesdropping and ARP spoofing.

Well known network-level threats include the following:

- *Eavesdropping* — Even when using cryptography, eavesdropping may be possible when not performing proper authentication, using a man-in-the-middle attack.
- Tampering — An attacker can change data on the wire. Even if the data is encrypted, it may be possible to make significant changes to the data without being able to decrypt it. Tampering is best thwarted by performing ongoing message authentication (MACing), provided by most high-level protocols, such as SSL/TLS.
- *Spoofing* — Traffic can be forged so that it appears to come from a different source address than the one from which it actually comes. This will thwart authentication systems that rely exclusively on IP addresses and/or DNS names for authentication.
- *Hijacking* — An extension of spoofing, established connections can be taken over, allowing the attacker to enter an already established session without having to authenticate. This can be thwarted with ongoing message authentication, which is provided by most high-level protocols, such as SSL/TLS.
- *Observing* — It is possible to give away security-critical information even when a network connection is confidentiality-protected through encryption. For example, the mere fact that two particular hosts are talking may give away significant information, as can the timing of traffic. These are generally examples of covert channels (non-obvious communication paths), which tend to be the most difficult problem in the security space.

Minimize Attack Surface

For a large application, a rough yet reliable metric for determining overall risk is to measure the number of input points that the application has — i.e., *attack surface*. The notion is that more points of entry into the application provides more avenues for an attacker to find a weakness.

Of course, any such metric must consider the accessibility of the input point. For example, many applications are developed for a threat model where the local environment is trusted. In this case, having a large number of local input points such as configuration files, registry keys, user input, etc., should be considered far less worrisome than making several external network connections.

Collapsing functionality that previously was spread across several ports onto a single port does not always help reduce attack surface, particularly when the single port exports all the same functionality, with an infrastructure that performs basic switching. The effective attack surface is the same unless the actual functionality is somehow simplified. Since underlying complexity clearly plays a role, metrics based on attack surface should not be used as the only means access control should be mandatory of analyzing risks in a piece of software.

Secure-by-Default

A system's default setting should not expose users to unnecessary risks and should be as secure as possible. This means that all security functionality should be enabled by default, and all optional features which entail any security risk should be disabled by default.

It also means that — if there is some sort of failure in the system — the behavior should not cause the system to behave in an insecure manner (the "fail-safe" principle). For example, if a connection cannot be established over SSL, it is not a good idea to try to establish a plaintext connection.

The "secure-by-default" philosophy does not interact well with usability since it is far simpler for the user to make immediate use of a system if all functionality is enabled. He can make use of functionality which is needed and ignore the functionality that is not.

However, attackers will not ignore this functionality. A system released with an insecure default configuration ensures that the vast majority of systems-in-the-wild are vulnerable. In many circumstances, it can even become difficult to patch a system before it is compromised.

Therefore, if there are significant security risks that the user is not already accepting, you should prefer a secure-by-default configuration. If not, at least alert the user to the risks ahead of time and point him to documentation on mitigation strategies.

Note that, in a secure-by-default system, the user will have to explicitly enable any functionality that increases his risk. Such operations should be relatively hidden (e.g., in an "advanced" preference pane) and should make the risks in disabling the functionality readily apparent.

Defense-in-Depth

The principle of defense-in-depth is that redundant security mechanisms increase security. If one mechanism fails, perhaps the other one will still provide the necessary security. For example, it is not a good idea to rely on a firewall to provide security for an internal-use-only application, as firewalls can usually be circumvented by a determined attacker (even if it requires a physical attack or a social engineering attack of some sort).

Implementing a defense-in-depth strategy can add to the complexity of an application, which runs counter to the “simplicity” principle often practiced in security. That is, one could argue that new protection functionality adds additional complexity that might bring new risks with it.

The risks need to be weighed. For example, a second mechanism may make no sense when the first mechanism is believed to be 100% effective; therefore, there is not much reason for introducing the additional solution, which may pose new risks. But usually the risks in additional complexity are minimal compared to the risk the protection mechanism seeks to reduce.

Principles for Reducing Exposure

Submarines employ a trick that makes them far less risky to inhabit. Assume that you are underwater on a sub when the hull bursts right by you. You actually have a reasonable chance of survival, because the ship is broken up into separate airtight compartments. If one compartment takes on water, it can be sealed off from the rest of the compartments.

Compartmentalization is a good principle to keep in mind when designing software systems. The basic idea is to try to contain damage if something does goes wrong. Another principle is that of *least privilege*, which states that privileges granted to a user should be limited to only those privileges necessary to do what that user needs to do. For example, least privilege argues that you should not run your program with administrative privileges, if at all possible. Instead, you should run it as a lesser user with just enough privileges to do the job, and no more.

Another relevant principle is to minimize windows of vulnerability. This means that — when risks must be introduced — they should be introduced for as short a time as possible (a corollary of this is "insecure bootstrapping"). In the context of privilege, it is could to account for which privileges a user can obtain, but only grant them when the situation absolutely merits. That supports the least privilege principle by granting the user privileges only when necessary, and revoking them immediately after use.

When the resources you are mitigating access in order to live outside your application, these principles are usually easier to apply with operational controls than with controls you build into your own software. However, one highly effective technique for enforcing these principles is the notion of *privilege separation*. The idea is that an application is broken up into two portions, the privileged core and the main application. The privileged core has as little functionality as absolutely possible so that it can be well audited. Its only purposes are as follows:

- Authenticate new connections and spawn off unprivileged main processes to handle those connections.
- Mediate access to those resources which the unprivileged process might legitimately get to access. That is, the core listens to requests from the children, determines whether they are valid, and then executes them on behalf of the unprivileged process.

This technique compartmentalizes each user of the system into its own process and completely removes all access to privileges, except for those privileges absolutely necessary, and then grants those privileges indirectly, only at the point where it is necessary.

The Insecure-Bootstrapping Principle

Insecure bootstrapping is the principle that — if you need to use an insecure communication channel for anything — you should use it to bootstrap a secure communication channel so that you do not need to use an insecure channel again.

For example, SSH is a protocol that provides a secure channel after the client and server have authenticated each other. Since it does not use a public key infrastructure the first time the client connects, it generally will not have the server credentials. The server sends its credentials, and the client just blindly accepts that they're the right ones. Clearly, if an attacker can send his own credentials, he can masquerade as the server or launch a man-in-the-middle attack.

But, the SSH client remembers the credentials. If the credentials remain the same, and the first connection was secure, then subsequent connections are secure. If the credentials change, then something is wrong — i.e., either an attack is being waged, or the server credentials have changed — and SSH clients will generally alert the user.

Of course, it is better not to use an insecure communication channel at all, if it can be avoided.

B: Example of Basic Principle: Input Validation

If a program is liberal in what it accepts, it often risks an attacker finding an input that has negative security implications. Several major categories of software security problems are ultimately input validation problems — including buffer overflows, SQL injection attacks, and command-injection attacks.

Data input to a program is either valid or invalid. What defines valid can be dependent on the semantics of the program. Good security practice is to definitively identify all invalid data before any action on the data is taken. And, if data is invalid, one should act appropriately.

Where to perform input validation

There are many levels at which one can perform input validation. Common places include:

- *Use* — all places in the code where data (particularly data of external origin) gets used.
- *Unit boundaries* — i.e., individual components, modules, or functions;
- *Trust boundaries* — i.e., on a per-executable basis.
- *Protocol parsing* — When the network protocol gets interpreted.
- *Application entry points* — e.g., just before or just after passing data to an application, such as a validation engine in a web server for a web service.
- *Network* — i.e., a traditional intrusion detection system (IDS).

Validating at use is generally quite error-prone because it is easy to forget to insert a check. This is still true, but less so when validating at unit boundaries. Going up the line, validation becomes less error prone. However, at higher levels, it gets harder and harder to make accurate checks because there is less and less context readily available to make a decision with.

At a bare minimum, input validation should be performed at unit boundaries, preferably using a structured technique such as design-by-contract. Validating at other levels provides defense-in-depth to help handle the case where a check is forgotten at a lower level.

Ways in which data can be invalid

At a high level, invalid data is anything that does not meet the strictest possible definition of valid. It does not just encompass malformed data, it encompasses missing data and out-of-order data (e.g., data used in a capture-replay attack).

There are four different contexts in which data can be invalid:

- *Sender* — Data is invalid if it did not originate from an authentic source.
- *Tokens* — Data in network protocols are generally broken up into atomic units called tokens, which often map to concrete data types (e.g., numbers, zip codes, and strings). An invalid token is one that is an invalid value for all token types known to a system.
- *Syntax* — Protocols accept messages as valid based on a protocol syntax, which is usually defined in terms of tokens. An invalid message is one that should not be accepted as part of the protocol.
- *Semantics* — Even when a message satisfies syntax requirements, it may be semantically invalid.

How to determine input validity

Data validity must be evaluated in each of the four contexts described above.   For example, a valid sender can send bad tokens. Good tokens can be combined in syntactically invalid ways. And, otherwise valid messages can make no valid sense in terms of the program's semantics.

At a high-level, there are three approaches to providing data validity:

- *Black-listing* — Widely considered bad practice in all cases, one validates based on a policy that explicitly defines bad values. All other data is assumed to be valid, but in practice, it often is not (or should not be).
- *White-listing* — One validates based on a precise description of what valid data entails (a policy). If the policy is correct, this prevents accidentally allowing maliciously invalid data. The risks are that the policy will not be correct, which may result not only in allowing bad data but also in disallowing some valid data.
- *Cryptographic validation* — One uses cryptography to demonstrate validity of the data.

Handling each input validation context involves a separate strategy:

- The sender can, in the general case, only be validated adequately using cryptographic message authentication.
- Tokens are generally validated using a simple state machine describing valid tokens (often implemented with regular expressions).
- Syntax is generally validated using a standard language parser, such as a recursive decent parser or a parser generated by a parser generator.
- Semantics are generally validated at the highest boundary at which all of the semantic data needed to make a decision is available. Message-ordering omission is best validated cryptographically along with sender authentication.

Protocol-specific semantics are often best validated in the context of a parser generated from a specification. In this case, semantics should be validated in the production associated with a single syntactic rule. When not enough semantic data is available at this level, semantic validation is best performed using a design-by-contract approach.

Actions to perform when invalid data is found

There are three classes of action one can take when invalid data is identified:

- *Error* — This includes fatal errors and non-fatal errors.
- *Record* — This includes logging errors and sending notifications of errors to interested parties.
- *Modify* — This includes filtering data or replacing data with default values.

These three classes are orthogonal, meaning that the decision to do any one is independent from the others. One can easily perform all three classes of action.

C: Example of Basic-Principle Violation: Penetrate-and-Patch Model

Addressing the application-security problem effectively is difficult because traditional software development lifecycles do not deal with these concerns well. This is largely due to a lack of structured guidance, since the few books on the topic are relatively new and document only collections of best practices.

In addition, security is not a feature that demonstrates well. Development organizations generally prefer to focus on core functionality features and then address security in an *ad-hoc* manner during development — focusing on providing a minimal set of services and driven by the limited security expertise of developers. This usually results in over-reliance on poorly understood security technologies.

Example: SSL Deployments

SSL is the most popular means of providing data confidentiality and integrity services for data traversing a network. Yet most SSL deployments are susceptible to network-based attacks because the technology is widely misunderstood by those who apply it. Particularly, people tend to treat it as a drop-in for traditional sockets, but when used in this way necessary server authentication steps are skipped. Performing proper authentication is usually a highly complex process.

Organizations that deploy technologies such as SSL and Java are often susceptible to a false sense of security. For example, an informal study of Java programs was conducted, which showed that a significant security risk appeared, on average, once per thousand lines of code — an extremely high number.

Cost of Deferring Security Issues

The result of the traditional shoestring approach to software security is that organizations will cross their fingers, hoping that security problems do not manifest and deferring most security issues to the time when they do — which is often well after the software is deployed. This is the so-called “penetrate-and-patch” model.

Bolting on a security solution after a problem is found is, of course, just as nonsensical as adding on a reliability module to fix robustness problems after software is developed. In fact, an IBM study on the cost of addressing security issues at various points during the SDLC argues that the cost of deferring security issues from design into deployment is greater than the cost associated with traditional reliability bugs. This is largely due to the tremendous overhead associated with vulnerability disclosure and actual security breaches.

D: Core Security Services

There are several fundamental security goals that may be required for the resources in your system. For each resource in your system, you should be aware of whether and how you are addressing each concern throughout the lifetime of the resource. That is, each resource may have different protection requirements as it interacts with different resources. For example, user data may not need to be protected on the user's machine but may need long-term secure storage in your database to prevent against possible insider attacks.

The fundamental security goals are: access control, authentication, confidentiality, data integrity, availability, accountability, and non-repudiation. In this section, we give an overview of each of the goals, explaining important nuances and discussing the levels within a system at which the concern can be addressed effectively.

Be aware that mechanisms put in place to achieve each of these services may be thwarted by unintentional logic errors in code.

Authorization (access control)

Authorization — also known as access control — is mediating access to resources on the basis of identity and is generally policy-driven (although the policy may be implicit). It is the primary security service that concerns most software, with most of the other security services supporting it. For example, access control decisions are generally enforced on the basis of a user-specific policy, and authentication is the way to establish the user in question. Similarly, confidentiality is really a manifestation of access control, specifically the ability to read data. Since, in computer security, confidentiality is often synonymous with encryption, it becomes a technique for enforcing an access-control policy.

Policies that are to be enforced by an access-control mechanism generally operate on sets of resources; the policy may differ for individual actions that may be performed on those resources (capabilities). For example, common capabilities for a file on a file system are: read, write, execute, create, and delete. However, there are other operations that could be considered “meta-operations” that are often overlooked — particularly reading and writing file attributes, setting file ownership, and establishing access control policy to any of these operations.

Often, resources are overlooked when implementing access control systems. For example, buffer overflows are a failure in enforcing write-access on specific areas of memory. Often, a buffer overflow exploit also accesses the CPU in a manner that is implicitly unauthorized as well.

Advantage of Mandatory Access Control

From the perspective of end-users of a system, access control should be mandatory whenever possible, as opposed to discretionary. Mandatory access control means that the system establishes and enforces a policy for user data, and the user does not get to make his own decisions of who else in the system can access data. In discretionary access control, the user can make such decisions. Enforcing a conservative mandatory access control policy can help prevent operational security errors, where the end user does not understand the implications of granting particular privileges. It usually keeps the system simpler as well.

Mandatory access control is also worth considering at the OS level, where the OS labels data going into an application and enforces an externally defined access control policy whenever the application attempts to access system resources. While such technologies are only applicable in a few environments, they are particularly useful as a compartmentalization mechanism, since — if a particular application gets compromised — a good MAC system will prevent it from doing much damage to other applications running on the same machine.

Authentication

In most cases, one wants to establish the identity of either a communications partner or the owner, creator, etc. of data. For network connections, it is important to perform authentication at login time, but it is also important to perform ongoing authentication over the lifetime of the connection; this can easily be done on a per-message basis without inconveniencing the user. This is often thought of as message integrity, but in most contexts integrity is a side-effect of necessary re-authentication.

Authentication is a prerequisite for making policy-based access control decisions, since most systems have policies that differ, based on identity.

In reality, authentication rarely establishes identity with absolute certainty. In most cases, one is authenticating credentials that one expects to be unique to the entity, such as a password or a hardware token. But those credentials can be compromised. And in some cases (particularly in biometrics), the decision may be based on a metric that has a significant error rate.

Additionally, for data communications, an initial authentication provides assurance at the time the authentication completes, but when the initial authentication is used to establish authenticity of data through the life of the connection, the assurance level generally goes down as time goes on. That is, authentication data may not be "fresh," such as when the valid user wanders off to eat lunch, and some other user sits down at the terminal.

In data communication, authentication is often combined with key exchange. This combination is advantageous since there should be no unauthenticated messages (including key exchange messages) and since general-purpose data communication often requires a key to be exchanged. Even when using public key cryptography where no key needs to be exchanged, it is generally wise to exchange them because general-purpose encryption using public keys has many pitfalls, efficiency being only one of them.

Authentication factors

There are many different techniques (or factors) for performing authentication. Authentication factors are usually termed *strong* or *weak*. The term strong authentication factor usually implies reasonable cryptographic security levels, although the terms are often used imprecisely.

Authentication factors fall into these categories:

- *Things you know* — such as passwords or pass-phrases. These are usually considered weak authentication factors, but that is not always the case (such as when using a strong password protocol such as SRP and a large, randomly generated secret). The big problem with this kind of mechanism is the limited memory of users. Strong secrets are difficult to remember, so people tend to share authentication credentials across systems, reducing the overall security.

  Sometimes people will take a strong secret and convert it into a "thing you have" by writing it down. This can lead to more secure systems by ameliorating the typical problems with weak passwords; but it introduces new attack vectors.

- *Things you have* — such as a credit card or an RSA SecurID (often referred to as authentication tokens). One risk common to all such authentication mechanisms is token theft. In most cases, the token may be cloneable. In some cases, the token may be used in a way that the actual physical presence is not required (e.g., online use of credit card doesn't require the physical card).

- *Things you are* — referring particularly to biometrics, such as fingerprint, voiceprint, and retinal scans. In many cases, readers can be fooled or circumvented, which provides captured data without actually capturing the data from a living being.

A system can support multiple authentication mechanisms. If only one of a set of authentication mechanisms is required, the security of the system will generally be diminished, as the attacker can go after the weakest of all supported methods.

However, if multiple authentication mechanisms must be satisfied to authenticate, the security increases (the defense-in-depth principle). This is a best practice for authentication and is commonly called *multi-factor authentication*. Most commonly, this combines multiple kinds of authentication mechanism — such as using both SecurID cards and a short PIN or password.

Who is authenticated?

In a two-party authentication (by far, the most common case), one may perform one-way authentication or mutual authentication. In one-way authentication, the result is that one party has confidence in the identity of the other — but not the other way around. There may still be a secure channel created as a result (i.e., there may still be a key exchange).

Mutual authentication cannot be achieved simply with two parallel one-way authentications, or even two one-way authentications over an insecure medium. Instead, one must cryptographically tie the two authentications together to prove there is no attacker involved.

A common case of this is using SSL/TLS certificates to validate a server without doing a client-side authentication. During the server validation, the protocol performs a key exchange, leaving a secure channel, where the client knows the identity of the server — if everything was done properly. Then the server can use the secure channel to establish the identity of the client, perhaps using a simple password protocol. This is a sufficient proof to the server as long as the server does not believe that the client would intentionally introduce a proxy, in which case it may not be sufficient.

Authentication channels

Authentication decisions may not be made at the point where authentication data is collected. Instead it may be proxied to some other device where a decision may be made. In some cases, the proxying of data will be non-obvious. For example, in a standard client-server application, it is clear that the client will need to send some sort of authentication information to the server. However, the server may proxy the decision to a third party, allowing for centralized management of accounts over a large number of resources.

It is important to recognize that the channel over which authentication occurs provides necessary security services. For example, it is common to perform password authentication over the Internet in the clear. If the password authentication is not strong (i.e., a zero-knowledge password protocol), it will leak information, generally making it easy for the attacker to recover the password. If there is data that could possibly be leaked over the channel, it could be compromised.

Confidentiality

It is often a requirement that data should be secret to all unauthorized parties, both when in transit on a network and when being stored, long-term or short-term.

Confidentiality is often synonymous with encryption, but there is more to confidentiality than merely encrypting data in transit or in storage. For example, users may have privacy requirements relative to other users, where systems that use encryption alone will often behave improperly. In particular, in a system with multiple users — where each user will want to allow some subset of other users to see the data, but not others — good mediation is mandatory. Otherwise, a server that mistakenly ships off data against the wishes of a customer is likely to encrypt the data but to the wrong entity.

Additionally, confidentiality can be compromised even when properly mediating access between resources and performing encryption. Potential attackers may be able to learn important information simply by observing the data you send. As a simple example, consider a system where Bob asks Alice questions so that everyone knows in advance, and Alice simply responds “yes” or “no” to each of them.

If Alice’s responses each go out in a single packet, and each answer is encoded in text (particularly, “yes” and “no”) instead of a single bit, then an attacker can determine the original plaintext without breaking the encryption algorithm simply by monitoring the size of each packet. Even if all of the responses are sent in a single packet, clumped together, the attacker can at least determine how many responses are “yes” and how many are “no” by measuring the length of the string.

Example: Assume that there are twenty questions, and the ciphertext is 55 characters. If every answer were “no”, then the ciphertext would be 40 characters long. Since there are 15 extra characters, and “yes” is one character longer than “no,” there must have been 15 “yes” answers.

Lapses in confidentiality such as this one that are neither obvious nor protected by standard encryption mechanisms are called “covert channels.” Another case of a covert channel occurs when the attacker can gain information simply by knowing which parties are talking to each other. There, he can often tell by monitoring the encrypted packets on the wire which have destination addresses. Even when the destination addresses are encrypted, the attacker may be able to observe the two endpoints and correlate a particular amount of traffic leaving one location with the same amount of traffic arriving at another location at the same time.

Covert channels are generally classified as either covert-storage channels or covert-timing channels. The previous example is a classic covert-timing channel. In covert-storage channels, artifacts of the way data is represented can communicate information, much like in our “yes” and “no” example. Also, when there are multiple ways of encoding the same information that are valid, it may be possible for two users to communicate additional unauthorized data by choosing a particular encoding scheme. This may be a concern, depending on the application. For example, in an on-line game, it may give two parties a way to communicate detailed data that would constitute cheating and would not be easy to communicate via other mechanisms; particularly, if the data is complex data such as game coordinates and is inserted and removed automatically; reading such things over the phone in a timely manner may be impossible.

Data Integrity

In communications and data storage, it is almost always desirable to know that data is in the form it was intended to be. Data integrity checking allows one to make that determination. This generally implies authentication because the mechanism for determining that data has not been modified requires a secret possessed by the person who created the data. Proving the data has not changed in such a case is all done in the same operation as proving that the data originated with a particular sender.

For this reason, CLASP treats data integrity as a subset of data authentication. There are cases where integrity may be a separate service as authentication — such as at the physical link layer on trusted media, where errors may happen naturally but will not be security errors. These situations are extremely rare in software development.

Availability

Most systems that export resources, either directly or otherwise, come with some implicit understanding that those resources will generally be accessible (available). If an availability problem is caused maliciously, it is known as a denial of service attack.

Note that data delays can be considered an availability problem. For example, imagine sending a message that says, “sell 10 shares of MSFT” that an attacker delays until the price has plummeted to the point where the user would no longer want to sell those shares.

Accountability

Users of a system should generally be accountable for the actions they perform. In practice, this means that systems should log information on operations that could potentially require review. For example, financial transactions must always be tracked in order to abide by Sarbanes-Oxley regulations. For logs to be used in cases of accountability, they should generally be difficult to forge, using a message authentication scheme that protects the integrity of logs by authenticating the entity that performed the logging.

Non-Repudiation

In most two-party data communication, the two parties can prove to themselves whether data comes from an authentic source. But one generally does not have proof that a third party would find plausible. A message for which the original sender or some endorser can be established to third parties is said to be non-repudiatable. This security service is generally associated with digital signature schemes.

Note that legal systems do not have an absolute notion of non-repudiation. Particularly, in a court of law, “duress” is a valid way to repudiate a message. For example, Alice could sign a message to Bob that Bob uses against Alice in court, but Alice may have a legitimate duress defense if she was forced to send the message by someone holding a gun to her head.

E: Sample Coding Guidelines (Worksheets)

This resource contains sample coding guidelines in the form of templates and worksheets to support the CLASP process. These templates are meant to be tailored to the individual needs of your organization.

Note: For convenience, each worksheet can be pasted into a MS Word document.

Instructions to Manager

This worksheet is an example set of coding standards for a company performing software development. The guidelines are presented in table format, with a column left blank. The blank column is meant either for the implementor to take notes related to the guideline or for an auditor to determine whether the developer's work conforms to the coding guidelines.

Many of the guidelines in this worksheet are items that should be addressed at design time. We leave them in this guidelines document, both for those organizations that have not used CLASP during the design phase and for those cases where the implementor finds himself making design decisions.

We encourage you to remove those guidelines that do not apply to your organization since developers will be more prone to use the document if the number of irrelevant pieces are minimal.

Instructions to Developer

This worksheet enumerates standards for security that you are expected to follow in the course of implementation work. For each guideline, you should keep notes detailing where in the system the issue is relevant, along with the status of the guideline — e.g., steps that have been taken in the spirit of the guideline. Keeping track of this data can help independent security reviewers understand the security posture of the system much more quickly than they would be able to do otherwise.

If you believe that there are circumstances that would keep you from following one of these guidelines, seek approval of your manager.

E1: General Guidelines

• Do not use functionality that might call a command processor — e.g., system(), popen(), execp(), Perl's open()). Instead, use functionality that invokes programs without using a command shell — e.g., execv(). • **Notes:**
• Specify preconditions and post-conditions for each parameter and any fields or global variables used. • **Notes:**
• Initialize all variables on allocation. • **Notes:**
• Do not place sensitive data in containers that are difficult or impossible to erase securely (e.g., Strings in Java). • **Notes:**

• Erase all sensitive data immediately upon use — including moving from one memory location to another. Do not rely on a garbage collector to do this for you. • **Notes:**
• Do not open files as a privileged user. Instead, use other identities to compartmentalize. • **Notes:**
• For any function that can potentially return an error code (even if through a reference parameter), check the return value and handle it appropriately. • **Notes:**
• Log logins, file access, privilege elevation, and any financial transactions. • **Notes:**

• Do not use elevated privilege unless absolutely necessary — e.g., privileged blocks in Java or setuid in C/C++.   • **Notes:**
• When writing privileged code, drop privileges as quickly as possible.   • **Notes:**
• Keep privileged code blocks as short and simple as possible.   • **Notes:**
• If random numbers are necessary, use system-level high-quality randomness.   • **Notes:**

• Minimize calls to other languages, and ensure that calls to other languages do not subvert security checks in the system. • **Notes:**
• Do not store security-critical data in client-side code. • **Notes:**
• Perform code signing on all external software releases, public or private. • **Notes:**
• Do not use functionality that might call a command processor — e.g., system(), popen(), execp(), Perl's open()). Instead, use functionality that invokes programs without using a command shell — e.g., execv(). • **Notes:**

E2: Build and Test

• Always compile with all reasonable warnings enabled and fix any warnings — whether or not they indicate a significant problem. • **Notes:**
• Run audit tools on a daily basis and follow any recommendations identified. • **Notes:**
• Use a generic lint tool on a daily basis to supplement compiler warnings. • **Notes:**

E3: Network Usage

• Do not use TCP/IP sockets over loopback. • **Notes:**
• Use thread pools for handling network connections instead of generating one thread per connection. • **Notes:**
• Ensure all network connections are protected with confidentiality, integrity, and authentication mechanisms (including database connections). • **Notes:**
• For database connections, implement user-based access control via a “WHERE” clause. • **Notes:**

E4: Authentication

• When using SSL, ensure that the server identity is established by following a trust chain to a known root certificate. • **Notes:**
• When using SSL, validate the host information of the server certificate. • **Notes:**
• If weak client authentication is unavoidable, perform it only over a secure channel. • **Notes:**
• Do not rely upon IP numbers or DNS names in establishing identity. • **Notes:**

• Use strong password-based algorithms when possible — such as SRP. • **Notes:**
• Provide a mechanism for self-reset and do not allow for third-party reset. • **Notes:**
• Do not store passwords under any circumstances. Instead, use a cryptographically strong algorithm such as MD5-MCF to protect passwords. • **Notes:**
• Rate limit bad password guesses to 10 in a 5-minute period. • **Notes:**

• Provide a mechanism for users to check the quality of passwords when they set or change it. • **Notes:**
• Provide a mechanism for enforced password expiration that is configurable by the customer. • **Notes:**
• Avoid sending authentication information through E-mail, particularly for existing users. • **Notes:**

E5: Input Validation

• Perform input validation at all input entry points. • **Notes:**
• Perform input validation on any environment variables that are used. • **Notes:**
• Perform input validation at all entry points to modules. • **Notes:**
• Use prepared statements for database access. • **Notes:**

• Build accessor APIs to validate requests and to help enforce access control properties for any sensitive variables. • **Notes:**
• When converting data into a data structure (deserializing), perform explicit validation for all fields, ensuring that the entire object is semantically valid. • **Notes:**
• Do not allow spaces or special characters in user names. • **Notes:**
• Evaluate any URL encodings before trying to use the URL. • **Notes:**

• Validate all E-mail addresses, allowing only basic values. • **Notes:**
• Do not allow arbitrary HTML in items that may possibly be displayed on a web page. • **Notes:**
• Detect illegal UTF8 sequences. • **Notes:**

E6: File System

• Validate all filenames and directories before use, ensuring that there are no special characters that might lead to accessing an unintended file. • **Notes:**
• Use “safe directories” for all file access except those initiated by the end user — e.g., document saving and restoring to a user-chosen location. • **Notes:**
• Validate the safety of file system accesses atomically whenever used. • **Notes:**
• Have at least 64 bits of randomness in all temporary file names. • **Notes:**

E7: Documentation

• For all of the input validation points in the program, specify valid input space in documentation and comments. • **Notes:**
• Document any operational assumptions made by the software. • **Notes:**

E8: Object-Oriented Programming

• Specify class invariants for each field. If no support for run-time invariant checking is available, include invariant specifications in the class comments.   • **Notes:**
• Do not use public variables — use accessors instead (particularly in mobile code/untrusted environments).   • **Notes:**

E9: Cryptography

• All protocols and algorithms for authentication and secure communication should be well vetted by the cryptographic community. • **Notes:**
• Do not use stream ciphers for encryption. • **Notes:**
• Perform Message integrity checking by using a “combined mode of operation,” or a MAC based on a block cipher. • **Notes:**
• Do not use key sizes less than 128 bits or cryptographic hash functions with output sizes less than 160 bits. • **Notes:**

E10: UNIX-Specific

• Do not use the same signal handler to handle multiple signals. • **Notes:**
• Do not do I/O or call complex functionality from a signal handler. • **Notes:**

E11: Windows-Specific

- Do not use Windows user-interface APIs for windows (even invisible ones) and message loops from services running with elevated privileges.
- **Notes:**

E12: C, C++, Perl, Python, PHP

- Avoid use of any functions that are in the RATS database.
- **Notes:**

E13: C and C++

- Do not omit types or explicitly circumvent the type system with liberal use of void * — use the type checker to the utmost advantage.
- **Notes:**

- Use a reasonable abstraction for string handling — such as the standard string class in C++ or SafeStr in C.
- **Notes:**

- For formatted I/O functions, use static format strings defined at the call site.
- **Notes:**

- When deciding how much memory to allocate, check for wrap-around conditions and error if they occur.
- **Notes:**

• Check to see if memory allocation or reallocation fails; abort if it does. • **Notes:**
• Do not stack-allocate arrays or other large objects. • **Notes:**
• Do not create your own variable argument functions. • **Notes:**
• Be wary of multi-byte character functionality — such strings are twice as large as the number of characters, and sometimes larger. • **Notes:**

E14: Java Mobile Code

• Keep privileged code blocks private. • **Notes:**
• Do not use public static variables, unless also declared final. • **Notes:**
• Protect packages against class insertion attacks. • **Notes:**
• Use the transient keyword when serializing files, sockets, and other data that cannot survive a serialize. • **Notes:**

- Have classes define their own deserialization routines and have them validate under the assumption that an attacker has modified the input bytes.
- **Notes:**

E15: Web Applications

• Do not pass secret data in forms or URLs. • **Notes:**
• Do not pass secret data in cookies without having the server encrypt and integrity-protect the cookie first. • **Notes:**
• Ensure that session IDs are randomly chosen and contain an adequate security level (64 bits). • **Notes:**
• Do not trust the validity of the “Referrer” header or any other HTTP header. • **Notes:**

• Provide reasonable time-outs on sessions. • **Notes:**
• Ensure SSL protection for account creation and any financial transactions, with a publicly verifiable SSL certificate. • **Notes:**

E16: Generic Mobile / Untrusted Code Environments

• Do not pass around object references beyond class boundaries. Instead, do a deep copy of data structures when they are requested. • **Notes:**
• Only make methods public or protected when absolutely necessary. • **Notes:**

F: System Assessment (Worksheets)

This document is inspired in part by NIST Special Publication 800-26: *Security Self-Assessment Guide for Information Technology Systems*. This publication provides a more granular look at some of the management issues in development. While good for a self-assessment, it is a bit too detailed for many situations when dealing with a third-party vendor. In addition, it does not capture some information vital to CLASP.

Note: For convenience, each worksheet can be pasted into a MS Word document.

F1: Application Assessment Overview

A: Application Assessment Overview Worksheet

INITIATION DATE	
COMPLETION DATE	
APPLICATION NAME	
APPLICATION VERSION	
UNIQUE IDENTIFIER	
PURPOSE OF ASSESSMENT	
TRUST BOUNDARIES	
DESCRIPTION OF FUNCTIONALITY	
LIST OF COMPONENT SYSTEMS	

LIST OF CONNECTED SYSTEMS	
PRIMARY SYSTEM POC	
E-MAIL	
PHONE	
CITY, STATE, ZIP	
WWW	
OTHER SYSTEM POC	
E-MAIL	
PHONE	
CITY, STATE, ZIP	
WWW	

B: Assessment Results Overview

TO DO: *FILL THIS IN.*	

Please attach the following documentation to the system assessment, when possible:

- Architecture diagrams.
- Most recent complete assessment reports for design and implementation.
- Relevant secure coding guidelines.
- Operational security guide for the system.
- Any security documentation, such as architectural security document.

F2: System Assessment Cover Page

INITIATION DATE	
COMPLETION DATE	
SYSTEM NAME	
SYSTEM VERSION	
UNIQUE IDENTIFIER	
SYSTEM VENDOR	
TARGET SYSTEM OS(ES)	
TARGET SYSTEM PLATFORM(S)	
SYSTEM DESCRIPTION	
THIRD-PARTY DEPENDENCIES	

ROLES WITHIN SYSTEM	
BOUNDARY CONTROLS FOR CONNECTED COMPONENTS	
NOTES:	

F3: Development Process and Organization

Issue	Guidance	Solution
Are there periodic risk assessments of the system?		
Are risk assessments performed on the design?		
If so, who performs them?	Indicate team member, contractor, independent audit group.	
If so, what method is used?		

Issue	Guidance	Solution
Most recent architectural assessment (version and date)?	Please attach the most recent architectural assessment, and/or endorsement from third party, if applicable.	
Are risk assessments performed on the implementation?		
If so, who performs them?	Indicate team member, contractor, independent audit group. If contractor, specify firm.	
If so, what method is used?		

Issue	Guidance	Solution
Most recent implementation assessment (version and date)?	Please attach the most recent architectural assessment, and/or endorsement from third party, if applicable.	
Are automated tools used in implementation assessment?	Please specify yes or no, and the tool name(s), if yes.	
What version control and bug tracking systems does the team use for tracking security defects?		
Do you use a standard security awareness program for your development team?	If so, please attach curriculum, or provide an overview of topic areas covered.	

Issue	Guidance	Solution
If so, the duration of the program.		
how often do team members receive refreshers?		
Which teams have received training?	One or more of: Architect/ designers, developers, testers, managers.	
What percent of the product team has been through a security awareness program?		

Issue	Guidance	Solution
WHAT ACCOUNTABILITY MEASURES ARE IN PLACE FOR SECURITY FLAWS?		
DO YOU ENFORCE SECURE CODING STANDARDS?	If so, please attach standards, and detail how they are enforced within your organization.	
WHAT DISTRIBUTION MECHANISM(S) DO YOU USE FOR MAJOR SOFTWARE UPDATES?		
WHAT DISTRIBUTION MECHANISM(S) DO YOU USE FOR INCREMENTAL UPDATES?		

Issue	Guidance	Solution
WHAT SECURITY RISKS DEEMED ACCEPTABLE ARE PRESENT IN THE ASSESSED VERSION OF THE SYSTEM?		
DO YOU HAVE INTERNAL PROCESS FOR RESPONDING TO SECURITY INCIDENTS?		

Issue	Guidance	Solution
What is the maximum expected time from private disclosure to available fix?		
What is the maximum expected time from public disclosure to available fix?		
How do you notify customers of security incidents?		
What security risks have been found in your system previously?		

Issue	Guidance	Solution
Are there any outstanding security risks known to be in the system?	This should not include those risks that were explicitly deemed acceptable above.	
What is your corporate policy for product maintenance?	Particularly, specify the point where you will no longer support the product with security updates.	
What process do you use for security testing?	Please list relevant techniques used, including red teaming, fuzing, fault injection and dynamic web app testing.	
Does the system have available guidance for operational security?	If yes, please attach to this document.	

Issue	Guidance	Solution
Does your system provide mechanisms for data recovery or redundancy?		
What are the configurable security options in the system; what are their default settings?		
What user accounts are installed in the system by default, what is the default authentication process; how is this updated?		

F4: System Resources

In this section, list all of the distinct resources that this system uses internally or exports and denote measures taken to promote security goals, where appropriate.

Resource	Security Measures
	• Authentication: • Confidentiality: • Data integrity: • Access control: • Non-repudiation: • Accountability:
	• Authentication: • Confidentiality: • Data integrity: • Access control: • Non-repudiation: • Accountability:

Resource	Security Measures
	• Authentication: • Confidentiality: • Data integrity: • Access control: • Non-repudiation: • Accountability:
	• Authentication: • Confidentiality: • Data integrity: • Access control: • Non-repudiation: • Accountability:
	• Authentication: • Confidentiality: • Data integrity: • Access control: • Non-repudiation: • Accountability:

Resource	Security Measures
	• Authentication: • Confidentiality: • Data integrity: • Access control: • Non-repudiation: • Accountability:
	• Authentication: • Confidentiality: • Data integrity: • Access control: • Non-repudiation: • Accountability:
	• Authentication: • Confidentiality: • Data integrity: • Access control: • Non-repudiation: • Accountability:

F5: Network Resource Detail

On this page, specify the ports and protocols that are used by the system, denoting the individual resources that may be accessed or sent through that channel. Additionally, specify operational security assumptions — such as whether the port is expected to be behind a firewall, expected to communicate with only a particular piece of software, etc.

Port	Protocols	Resources	Notes:

Port	Protocols	Resources	Notes:

Port	Protocols	Resources	Notes:

Port	Protocols	Resources	Notes:

Port	Protocols	Resources	Notes:

F6: File System Usage Detail

In this section, detail which resources on the file system can be used by the program. For each file or directory, indicate the privileges needed (e.g., owner, administrator), the type of access required (read, write, execute, etc.), and an indication of whether the resource has special security measures taken for confidentiality, integrity, and other security services.

The data in this table can be used to establish a sandboxing or monitoring environment.

File or Directory	Privileges Needed	Type of Access	Security Measures

File or Directory	**Privileges Needed**	**Type of Access**	**Security Measures**

File or Directory	**Privileges Needed**	**Type of Access**	**Security Measures**

File or Directory	Privileges Needed	Type of Access	Security Measures

File or Directory	Privileges Needed	Type of Access	Security Measures

File or Directory	Privileges Needed	Type of Access	Security Measures

File or Directory	Privileges Needed	Type of Access	Security Measures

File or Directory	Privileges Needed	Type of Access	Security Measures

File or Directory	Privileges Needed	Type of Access	Security Measures

File or Directory	Privileges Needed	Type of Access	Security Measures

F7: Registry Usage (Microsoft Windows Environment)

For programs running in a Microsoft Windows environment, indicate registry resources that are used by the system, along with the owner, actions that may be taken on the key (read, write, delete, etc.), and notes on the security relevance of the key.

Registry Key	Owner	Type of Access	Notes: Registry Usage

Registry Key	Owner	Type of Access	Notes: Registry Usage

Registry Key	Owner	Type of Access	Notes: Registry Usage

Registry Key	Owner	Type of Access	Notes: Registry Usage

Registry Key	Owner	Type of Access	Notes: Registry Usage

Registry Key	Owner	Type of Access	Notes: Registry Usage

Registry Key	Owner	Type of Access	Notes: Registry Usage

Registry Key	Owner	Type of Access	Notes: Registry Usage

Registry Key	Owner	Type of Access	Notes: Registry Usage

Registry Key	Owner	Type of Access	Notes: Registry Usage

G: Sample Roadmaps

To help you navigate the CLASP activities more efficiently, we provide sample roadmaps which focus on common organizational requirements. There are two roadmaps:

- A Legacy application roadmap aimed at organizations looking for a minimal impact on their ongoing development projects, which introduces only those activities with the highest relative impact on security.
- A Green-Field roadmap that has been developed for organizations that are looking for a more holistic approach to application-security development practices. This roadmap is recommended for new software development, using a spiral or iterative methodology.

G1: Green-Field Roadmap

This green-field roadmap is for use by organizations that are looking for a more holistic approach to application-security development practices. This roadmap is recommended for new software development, using a spiral or iterative methodology.

Activity	Comments
• Institute security awareness program	
• Monitor security metrics	
• Specify operational environment	This step is important as a foundation for security analysis.
• Identify global security policy	
• Identify resources and trust boundaries	This step is also important as a foundation for security analysis.
• Identify user roles and resource capabilities	
• Document security-relevant requirements	Some attempt should be made to address resource-driven requirements from the system — both implicit and explicit — even if not to the level of depth as would be performed for Green Field development.
• Identify attack surface	This step is also important as a foundation for security analysis.
• Apply security principles to design	
• Research and assess security posture of technology solutions	
• Specify database security configuration	
• Perform security analysis of system requirements and design (threat modeling)	
• Integrate security analysis into source management process	
• Implement and elaborate resource policies and security technologies	
• Address reported security issues	
• Perform source-level security review	

Activity	Comments
• Identify, implement and perform security tests	
• Verify security attributes of resources	
• Build operational security guide	
• Manage security issue disclosure process	

G2: Legacy Roadmap

This legacy application roadmap aims at organizations looking for a minimal impact on their ongoing development projects, which introduces only those activities with the highest relative impact on security.

Activity	Comments
• Institute security awareness program	
• Specify operational environment	This step is important as a foundation for security analysis.
• Identify resources and trust boundaries	This step is also important as a foundation for security analysis.
• Document security-relevant requirements	Some attempt should be made to address resource-driven requirements from the system — both implicit and explicit — even if not to the level of depth as would be performed for Green Field development.
• Identify attack surface	This step is also important as a foundation for security analysis.
• Perform security analysis of system requirements and design (threat modeling)	
• Address reported security issues	
• Perform source-level security review	
• Identify, implement and perform security tests	
• Verify security attributes of resources	
• Build operational security guide	
• Manage security issue disclosure process	

H: Creating the Process Engineering Plan

To ensure an efficient ongoing process, it is important to carefully plan the process engineering effort. A good process engineering plan should include — at a minimum — the following elements:

- Business objectives that the process is being developed to meet.
- Project milestones and checkpoints.
- Pass/fail criteria for each milestone and checkpoint — e.g., necessary approvals, evaluation criteria, and stakeholder involvement.

Business objectives

While your team is documenting business objectives for an impending process engineering effort, bring into consideration any global application software development security policies that may already exist for the project or the organization. This should include any existing certification requirements.

Another objective at this point should be to agree on the set of security metrics that will be collected and monitored externally to the project throughout the process deployment phases in order to measure overall security posture. For example, security posture can be determined based on:

- Internal security metrics collected;
- Independent assessment (which can be performed using CLASP activities as well);
- Or — less desirably — through externally reported incidents involving the effort.

Process milestones

Your team should construct a draft process engineering plan, which identifies the key project milestones to be met for the project. The focus should be on when activities should be introduced, who should perform them, and how long they should take to perform.

Process evaluation criteria

As a final step in your planning efforts for process engineering, you should decide upon the criteria for measuring the success of your team, as well as the process engineering and deployment effort.

Success might be measured in one or more of many different methods, such as:

- Comparing the rate of deployment across projects;
- Comparing the percentage of security faults identified in development versus those found in production; or
- Monitoring the timeliness, accuracy, and thoroughness of key development artifacts.

Be specific, but be realistic in identifying success metrics. Remember that this process will evolve to meet your ever-changing and demanding business needs. Small successes early on will be more rewarding for the team than big failures, so consider a slow roll-out of new processes, with an accompanying incremental rollout of metrics.

I: Forming the Process Engineering Team

Development organizations should be bought into the process which they use for development. The most effective way to do that is to build a process engineering team from members of the development team so that they can have ownership in creating the process.

Steps to form Team

We recommend taking the following steps to form the process engineering team:

- Build a process engineering mission statement.

  Document the objectives of the process team. It is reasonable to have the entire development team sign off on the mission, so that those people who are not on the team still experience buy-in and inclusion.

- Identify a process owner.

  The process team should have a clearly identified process “champion,” whose foremost job is to set a direction and then evangelize that direction. Make it clear that this team will be held accountable for all aspects of the engineering and deployment activities associated with early adoption of this new security process framework.

- Identify additional contributors.

  As with the process owner, people who make good evangelists should be valued as well as people who will be the most worthy contributors.

- Document roles and responsibilities.

  Clearly document the roles and responsibilities of each member of this team.

- Document the CLASP process roadmap.

  It is time to make the classic “build-versus-buy” decision for a process framework. Can one of the process roadmaps packaged as part of CLASP be used as-is? Can the team simply extend one of the packaged roadmaps to meet either organizations software development needs? Does the team really need to step back and opportunistically chose discrete activities — thereby building a unique process framework that provides a “best fit” for their organization? This decision and the resulting process roadmap must be documented and approved before moving into the deployment phase. See the following section for sample roadmaps.

- Review and approve pre-deployment.

  Institute a checkpoint before deployment, in which a formal walk-through of the process is conducted. The objective at this point is to solicit early feedback on whether or not the documented framework will indeed meet the process objectives set forth at the beginning of this effort. The team should not proceed to the deployment phase of this project until organizational approval is formally issued.

- Document any issues.

  Issues that come up during the formation of the process engineering team should be carefully documented. These issues will need to be added to the process engineering or process deployment plans — as appropriate to managing risk accordingly.

J: Glossary of Security Terms

This glossary contains a list of terms relevant to application security. The terms in this glossary are not specific to material found in the CLASP process.

Term	Description
3DES	*See: Triple DES*
Access Control List	A list of credentials attached to a resource indicating whether or not the credentials have access to the resource.
ACL	Access Control List
Active attack	Any network-based attack other than simple eavesdropping — i.e., a passive attack).
Advanced Encryption Standard	A fast general-purpose block cipher standardized by NIST (the National Institute of Standards and Technology). The AES selection process was a multi-year competition, where Rijndael was the winning cipher.
AES	*See: Advanced Encryption Standard*
Anti-debugger	Referring to technology that detects or thwarts the use of a debugger on a piece of software.
Anti-tampering	Referring to technology that attempts to thwart the reverse engineering and patching of a piece of software in binary format.
Architectural security assessment	*See: Threat Model.*
ASN.1	Abstract Syntax Notation is a language for representing data objects. It is popular to use this in specifying cryptographic protocols, usually using DER (Distinguished Encoding Rules), which allows the data layout to be unambiguously specified. *See also: Distinguished Encoding Rules.*
Asymmetric cryptography	Cryptography involving public keys, as opposed to cryptography making use of shared secrets. *See also: Symmetric cryptography.*
Audit	In the context of security, a review of a system in order to validate the security of the system. Generally, this either refers to code auditing or reviewing audit logs. *See also: Audit log; code auditing.*
Audit log	Records that are kept for the purpose of later verifying that the security properties of a system have remained intact.

Authenticate- and-encrypt	When using a cipher to encrypt and a MAC to provide message integrity, this paradigm specifies that one authenticates the plaintext and encrypts the plain-text, possibly in parallel. This is not secure in the general case. *See also: Authenticate-then-encrypt; encrypt-then-authenticate.*
Authenticate-then-encrypt	When using a cipher to encrypt and a MAC to provide message integrity, this paradigm specifies that one authenticates the plaintext and then encrypts the plaintext concatenated with the MAC tag. This is not secure in the general case, but usually works well in practice. *See also: Authenticate-and-encrypt, Encrypt-then-authenticate.*
Authentication	The process of verifying identity, ownership, and/or authorization.
Backdoor	Malicious code inserted into a program for the purposes of providing the author covert access to machines running the program.
Base 64 encoding	A method for encoding binary data into printable ASCII strings. Every byte of output maps to six bits of input (minus possible padding bytes).
Big endian	Refers to machines representing words most significant byte first. While x86 machines do not use big endian byte ordering (instead using little endian), the PowerPC and SPARC architectures do. This is also network byte order. *See also: Little endian.*
Birthday attack	Take a function f() that seems to map an input to a random output of some fixed size (a pseudo-random function or PRF). A birthday attack is simply selecting random inputs for f() and checking to see if any previous values gave the same output. Statistically, if the output size is S bits, then one can find a collision in $2^{S/2}$ operations, on average.
Bit-flipping attack	In a stream cipher, flipping a bit in the ciphertext flips the corresponding bit in the plaintext. If using a message authentication code (MAC), such attacks are not practical.
Blacklist	When performing input validation, the set of items that — if matched — result in the input being considered invalid. If no invalid items are found, the result is valid. *See also: Whitelist.*
Blinding	A technique used to thwart timing attacks.
Block cipher	An encryption algorithm that maps inputs of size *n* to outputs of size *n* (*n* is called the block size). Data that is not a valid block size must somehow be pad-ded (generally by using an encryption mode). The same input always produces the same output. *See also: Stream cipher.*

Blowfish	A block cipher with 64-bit blocks and variable length keys, created by Bruce Schneier. This cipher is infamous for having slow key-setup times.
Brute-force attack	An attack on an encryption algorithm where the encryption key for a ciphertext is determined by trying to decrypt with every key until valid plaintext is obtained.
Buffer overflow	A buffer overflow is when you can put more data into a memory location than is allocated to hold that data. Languages like C and C++ that do no built-in bounds checking are susceptible to such problems. These problems are often security-critical.
CA	*See Certification Authority.*
Canary	A piece of data, the absence of which indicates a violation of a security policy. Several tools use a canary for preventing certain stack-smashing buffer overflow attacks. *See also: Buffer overflow; Stack smashing.*
Capture-replay attacks	When an attacker can capture data off the wire and replay it later without the bogus data being detected as bogus.
Carter Wegmen + Counter mode	A parallelizable and patent-free high-level encryption mode that provides both encryption and built-in message integrity.
CAST5	A block cipher with 64-bit blocks and key sizes up to 128 bits. It is patent- free, and generally considered sound, but modern algorithms with larger block sizes are generally preferred (e.g., AES). *See also: AES.*
CBC Mode	*See: Cipher Block Chaining mode.*
CBC-MAC	A simple construction for turning a block cipher into a message authentication code. It only is secure when all messages MAC'd with a single key are the same size. However, there are several variants that thwart this problem, the most important being OMAC. *See also: OMAC.*
CCM mode	*See: Counter mode + CBC-MAC.*
Certificate	A data object that binds information about a person or some other entity to a public key. The binding is generally done using a digital signature from a trusted third party (a certification authority).
Certificate Revocation List	A list published by a certification authority indicating which issued certificates should be considered invalid.
Certificate Signing Request	Data about an entity given to a certification authority. The authority will package the data into a certificate and sign the certificate if the data in the signing request is validated.

Certification Authority	An entity that manages digital certificates — i.e., issues and revokes. Verisign and InstantSSL are two well known CAs.
CFB mode	*See: Cipher Feedback mode.*
Chain responder	An OCSP responder that relays the results of querying another OCSP responder. *See also: OCSP.*
Choke point	In computer security, a place in a system where input is routed for the purposes of performing data validation. The implication is that there are few such places in a system and that all data must pass through one or more of the choke points. The idea is that funneling input through a small number of choke points makes it easier to ensure that input is properly validated. One potential concern is that poorly chosen choke points may not have enough information to perform input validation that is as accurate as possible.
chroot	A UNIX system call that sets the root directory for a process to any arbitrary directory. The idea is compartmentalization: Even if a process is compromised, it should not be able to see interesting parts of the file system beyond its own little world. There are some instances where chroot &quot;jails&quot; can be circumvented; it can be difficult to build proper operating environments to make chroot work well.
Cipher-Block Chaining mode	A block cipher mode that provides secrecy but not message integrity. Messages encrypted with this mode should have random initialization vectors.
Cipher Feedback mode	A mode that turns a block cipher into a stream cipher. This mode is safe only when used in particular configurations. Generally, CTR mode and OFB mode are used instead since both have better security bounds.
Ciphertext	The result of encrypting a message. *See: Plaintext.*
Ciphertext stealing mode	A block cipher mode of operation that is similar to CBC mode except that the final block is processed in such a way that the output is always the same length as the input. That is, this mode is similar to CBC mode but does not require padding. *See also: Cipher Block Chaining mode; Padding.*
Code auditing	Reviewing computer software for security problems. *See also: Audit.*
Code signing	Signing executable code to establish that it comes from a trustworthy vendor. The signature must be validated using a trusted third party in order to establish identity.

Compartmentalization	Separating a system into parts with distinct boundaries, using simple, well-defined interfaces. The basic idea is that of containment — i.e., if one part is compromised, perhaps the extent of the damage can be limited. *See also: Jail; Chroot.*
Context object	In a cryptographic library, a data object that holds the intermediate state associated with the cryptographic processing of a piece of data. For example, if incrementally hashing a string, a context object stores the internal state of the hash function necessary to process further data.
Counter mode	A parallelizable encryption mode that effectively turns a block cipher into a stream cipher. It is a popular component in authenticated encryption schemes due to its optimal security bounds and good performance characteristics.
Counter mode + CBC-MAC	An encryption mode that provides both message secrecy and integrity. It was the first such mode that was not covered by patent.
CRAM	A password-based authentication mechanism using a cryptographic hash function (usually MD5). It does not provide adequate protection against several common threats to password-based authentication systems. HTTP Digest Authentication is a somewhat better alternative; it is replacing CRAM in most places.
CRC	Cyclic Redundancy Check. A means of determining whether accidental transmission errors have occurred. Such algorithms are not cryptographically secure because attackers can often forge CRC values or even modify data maliciously in such a way that the CRC value does not change. Instead, one should use a strong, keyed message authentication code such as HMAC or OMAC. *See also: HMAC, Message Authentication Code; OMAC.*
Critical extensions	In an X.509 certificate, those extensions that must be recognized by any software processing the certificate. If a piece of software does not recognize an extension marked as critical, the software must regard the certificate as invalid.
CRL	*See: Certificate Revocation List.*
Cross-site scripting	A class of problems resulting from insufficient input validation where one user can add content to a web site that can be malicious when viewed by other users to the web site. For example, one might post to a message board that accepts arbitrary HTML and include a malicious code item.
Cryptanalysis	The science of breaking cryptographic algorithms.
Cryptographic hash function	A function that takes an input string of arbitrary length and produces a fixed-size output — where it is unfeasible to find two inputs that map to the same output, and it is unfeasible to learn anything about the input from the output.

Cryptographic randomness	Data produced by a cryptographic pseudo-random number generator. The probability of figuring out the internal state of the generator is related to the strength of the underlying cryptography — i.e., assuming the generator is seeded with enough entropy.
Cryptography	The science of providing secrecy, integrity, and non-repudiation for data.
CSR	*See: Certificate Signing Request.*
CSS	Cross-site scripting. Generally, however, this is abbreviated to XSS in order to avoid confusion with cascading style sheets. *See: Cross-site scripting.*
CTR mode	*See: Counter mode.*
CWC mode	*See: Carter Wegmen + Counter mode.*
DACL	Discretionary Access Control List. In a Windows ACL, a list that determines access rights to an object. *See also: Access Control List.*
Davies-Meyer	An algorithm for turning a block cipher into a cryptographic one-way hash function.
Default deny	A paradigm for access control and input validation where an action must explicitly be allowed. The idea behind this paradigm is that one should limit the possibilities for unexpected behavior by being strict, instead of lenient, with rules.
Defense-in-depth	A principle for building systems stating that multiple defensive mechanisms at different layers of a system are usually more secure than a single layer of defense. For example, when performing input validation, one might validate user data as it comes in and then also validate it before each use — just in case something was not caught, or the underlying components are linked against a different front end, etc.
DEK	Data encrypting key.
Delta CRLs	A variation of Certificate Revocation Lists that allows for incremental updating, as an effort to avoid frequently re-downloading a large amount of unchanged data. *See also: Certificate Revocation List.*
Denial of service attack	Any attack that affects the availability of a service. Reliability bugs that cause a service to crash or go into some sort of vegetative state are usually potential denial-of-service problems.

DES	The Data Encryption Standard. An encryption algorithm standardized by the US Government. The key length is too short, so this algorithm should be considered insecure. The effective key strength is 56 bits; the actual key size is 64 bits — 8 bits are wasted. However, there are variations such as Triple DES and DESX that increase security while also increasing the key size. *See also: Advanced Encryption Standard; Triple DES.*
DESX	An extended version of DES that increases the resistance to brute-force attack in a highly efficient way by increasing the key length. The extra key material is mixed into the encryption process, using XORs. This technique does not improve resistance to differential attacks, but such attacks are still generally considered unfeasible against DES. *See also: DES.*
Dictionary attack	An attack against a cryptographic system, using precomputating values to build a dictionary. For example, in a password system, one might keep a dictionary mapping ciphertext pairs in plaintext form to keys for a single plaintext that frequently occurs. A large enough key space can render this attack useless. In a password system, there are similar dictionary attacks, which are somewhat alleviated by salt. The end result is that the attacker — once he knows the salt — can do a “Crack”-style dictionary attack. Crack-style attacks can be avoided to some degree by making the password verifier computationally expensive to compute. Or select strong random passwords, or do not use a password-based system.
Differential cryptanalysis	A type of cryptographic attack where an attacker who can select related inputs learns information about the key from comparing the outputs. Modern ciphers of merit are designed in such a way as to thwart such attacks. Also note that such attacks generally require enough chosen plaintexts as to be considered unfeasible, even when there is a cipher that theoretically falls prey to such a problem.
Diffie-Hellman key exchange	A method for exchanging a secret key over an untrusted medium in such a way as to preserve the secrecy of the key. The two parties both contribute random data that factors into the final shared secret. The fundamental problem with this method is authenticating the party with whom you exchanged keys. The simple Diffie-Hellman protocol does not do that. One must also use some public-key authentication system such as DSA. *See also: DSA; Station-to-station protocol.*
Digest size	The output size for a hash function.

Digital signature	Data that proves that a document (or other piece of data) was not modified since being processed by a particular entity. Generally, what this really means is that — if someone 'signs' a piece of data — anyone who has the right public key can demonstrated which private key was used to sign the data.
Digital Signature Algorithm	*See: DSA.*
Distinguished Encoding Rules	A set of rules used that describes how to encode ASN.1 data objects unambiguously. *See also: ASN.1.*
Distinguished Name	In an X.509 certificate, a field that uniquely specifies the user or group to which the certificate is bound. Usually, the Distinguished Name will contain a user's name or User ID, an organizational name, and a country designation. For a server certificate, it will often contain the DNS name of the machine.
DN	*See: Distinguished Name.*
DoS	Denial of Service. *See also: Denial of service attack.*
DSA	The Digital Signature Algorithm, a public key algorithm dedicated to digital signatures which was standardized by NIST. It is based on the same mathematical principles as Diffie-Hellman.
Eavesdropping attack	Any attack on a data connection where one simply records or views data instead of tampering with the connection.
ECB Mode	*See: Electronic Code Book mode.*
ECC	*See: Eliptic Curve Cryptography.*
EGD	*See: Entropy Gathering Daemon.*
Electronic Code Book mode	An encryption mode for block ciphers that is more or less a direct use of the underlying block cipher. The only difference is that a message is padded out to a multiple of the block length. This mode should not be used under any circumstances.
Eliptic Curve Cryptography	A type of public key cryptography that — due to smaller key sizes — tends to be more efficient that standard cryptography. The basic algorithms are essentially the same, except that the operations are performed over different mathematical groups (called eliptic curves).
EME-OAEP padding	A padding scheme for public key cryptography that uses a "random" value generated, using a cryptographic hash function in order to prevent particular types of attacks against RSA. *See also: PKCS #1 padding.*

Encrypt-then-authenticate	When using a cipher to encrypt and a MAC to provide message integrity, this paradigm specifies that one encrypts the plaintext, then MACs the ciphertext. This paradigm has theoretically appealing properties and is recommended to use in practice. *See also: Authenticate-and-encrypt; Authenticate-then-encrypt.*
Endianess	The byte ordering scheme that a machine uses (usually either little endian or big endian). *See also: Big endian; Little endian.*
Entropy	Refers to the inherent unknowability of data to external observers. If a bit is just as likely to be a 1 as a 0 and a user does not know which it is, then the bit contains one bit of entropy.
Entropy Gathering Daemon	A substitute for /dev/random; a tool used for entropy harvesting.
Entropy harvester	A piece of software responsible for gathering entropy from a machine and distilling it into small pieces of high entropy data. Often an entropy harvester will produce a seed for a cryptographic pseudo-random number generator. *See also: Entropy; Pseudo-random number generator.*
Ephemeral keying	Using one-time public key pairs for session key exchange in order to prevent recovering previous session keys if a private key is compromised. Long-term public key pairs are still used to establish identity.
Euclidian algorithm	An algorithm that computes the greatest common divisor of any two numbers.
Extended Euclidian algorithm	An algorithm used to compute the inverse of a number modulo "some other number."
Fingerprint	The output of a cryptographic hash function. *See also: Message digest.*
FIPS	Federal Information Processing Standard; a set of standards from NIST.
FIPS-140	A standard authored by the U.S. National Institute of Standards and Technology, that details general security requirements for cryptographic software deployed in a government systems (primarily cryptographic providers). *See also: NIST; FIPS.*
Format string attack	The C standard library uses specifiers to format output. If an attacker can control the input to such a format string, he can often write to arbitrary memory locations.

Forward secrecy	Ensuring that the compromise of a secret does not divulge information that could lead to data protected prior to the compromise. In many systems with forward secrecy, it is only provided on a per-session basis, meaning that a key compromise will not affect previous sessions, but would allow an attacker to decrypt previous messages sent as a part of the current session. *See also: Perfect forward secrecy.*
Hash function	A function that maps a string of arbitrary length to a fixed size value in a deterministic manner. Such a function may or may not have cryptographic applications. *See also: Cryptographic hash function; Universal hash function; One-way hash function.*
Hash function (cryptographic)	*See: Cryptographic hash function.*
Hash function (one-way)	*See: One-way hash function.*
Hash function (universal)	*See: Universal hash function.*
Hash output	*See: Hash value.*
Hash value	The output of a hash function. *See also: Fingerprint; Message digest.*
hash127	A fast universal hash function from Dan Bernstein.
HMAC	A well-known algorithm for converting a cryptographic one-way hash function into a message authentication code.
IDEA	A block cipher with 128-bit keys and 64-bit blocks popularly used with PGP. It is currently protected by patents.
Identity establishment	Authentication.
IEEE P1363	An IEEE standard for eliptic curve cryptography. Implementing the standard requires licensing patents from Certicom.
Indirect CRLs	A CRL issued by a third party, that can contain certificates from multiple CA's. *See also: Certificate, Certificate Revocation List; Certification Authority.*
Initialization vector	A value used to initialize a cryptographic algorithm. Often, the implication is that the value must be random. *See also: Nonce; Salt.*
Input validation	The act of determining that data input to a program is sound.

Integer overflow	When an integer value is too big to be held by its associated data type, the results can often be disastrous. This is often a problem when converting unsigned numbers to signed values.
Integrity checking	The act of checking whether a message has been modified either maliciously or by accident. Cryptographically strong message integrity algorithms should always be used when integrity is important.
Interleaved encryption	Processing the encryption of a message as multiple messages, generally treat-ing every nth block as part of a single message.
IV	*See: Initialization vector.*
Jail	A restricted execution environment meant to compartmentalize a process, so that — even if it has security problems — it cannot hurt resources which it would not normally have access to use. On FreeBSD, a system call similar to chroot that provides compartmentalization. Unlike chroot, it can also restrict network resources in addition to file system resources. *See also: Chroot.*
Kerberos	“An authentication protocol that relies solely on symmetric cryptography, as opposed to public key cryptography. It still relies on a trusted third party (an authentication server). While Kerberos is often looked upon as a way to avoid problems with Public Key Infrastructure, it can be difficult to scale Kerberos beyond medium-sized organizations. *See also: Public Key Infrastructure; Trusted third party.*
Key agreement	The process of two parties agreeing on a shared secret, where both parties contribute material to the key.
Key establishment	The process of agreeing on a shared secret, where both parties contribute material to the key.
Key exchange	The process of two parties agreeing on a shared secret, usually implying that both parties contribute to the key.
Key management	Mechanisms and process for secure creation, storage, and handling of key material.
Key schedule	In a block cipher, keys used for individual “rounds” of encryption, derived from the base key in a cipher-dependent manner.
Key transport	When one party picks a session key and communicates it to a second party.
Keystream	Output from a stream cipher. *See also: Pseudo-random number generator; Stream cipher.*
LDAP	Lightweight Directory Access Protocol. A directory protocol commonly used for storing and distributing CRLs.

Length extension attack	A class of attack on message authentication codes, where a tag can be forged without the key by extending a pre-existing message in a particular way. CBC-MAC in its simplest form has this problem, but variants protect against it (particularly OMAC). *See also: Message Authentication Code; OMAC.*
LFSR	See: *Linear feedback shift register.*
Linear cryptanalysis	A type of cryptanalytic attack where linear approximations of behavior are used. Modern ciphers of merit are designed in such a way as to thwart such attacks. Also note that such attacks generally require enough chosen plaintexts as to be considered unfeasible — even when there is a cipher that theoretically falls prey to such a problem (such as DES).
Linear Feedback Shift Register	A non-cryptographic class of pseudo-random number generators, where output is determined by shifting out &quot;output&quot; bits and shifting in &quot;input&quot; bits, where the input bits are a function of the internal state of the register, perhaps combined with new entropy. LFSRs are based on polynomial math, and are not secure in and of themselves; however, they can be put to good use as a component in more secure cryptosystems.
Little endian	Refers to machines representing words of data least significant byte first, such as the Intel x86. *See also: Big endian.*
MAC	*See: Message authentication code.*
Man-in-the- middle attack	An eavesdropping attack where a client's communication with a server is proxied by an attacker. Generally, the implication is that the client performs a cryptographic key exchange with an entity and fails to authenticate that entity, thus allowing an attacker to look like a valid server.
Matyas-Meyer- Oseas	A construction for turning a block cipher into a cryptographic one-way hash function.
MCF	The Modular Crypt Format, a de-facto data format standard for storing password hashes commonly used on UNIX boxes as a replacement for the traditional UNIX crypt() format.
MD-strengthening	Merkel-Damgard strengthening, a general method for turning a collision- resistant compression function into a collision-resistant hash function by adding padding and an encoded length to the end of the input message. The key point behind MD-strengthening is that no possible input to the underlying hash function can be the tail end of a different input.
MD2	A cryptographic hash function optimized for 16-bit platforms. It has poor performance characteristics on other platforms and has a weak internal structure.

MD4	A cryptographic hash function that is known to be broken and should not be used under any circumstances.
MD5	A popular and fast cryptographic hash function that outputs 128-bit message digests. Its internal structure is known to be weak and should be avoided if at all possible.
MD5-MCF	A way of using MD5 to store password authentication information, using the modular crypt format. *See also: MCF, MD5.*
MDC2	A construction for turning a block cipher into a cryptographic hash function, where the output length is twice the block size of the cipher.
Meet-in-the- middle attack	A theoretical attack against encrypting a message twice using a single block cipher and two different keys. For example, double encryption with DES theoretically is no more secure than DES, which is why Triple DES became popular (it gives twice the effective key strength).
Message Authentication Code	A function that takes a message and a secret key (and possibly a nonce) and produces an output that cannot, in practice, be forged without possessing the secret key.
Message digest	The output of a hash function.
Message integrity	A message has integrity if it maintains the value it is supposed to maintain, as opposed to being modified on accident or as part of an attack.
Miller-Rabin	A primality test that is efficient because it is probabilistic, meaning that there is some chance it reports a composite (non-prime) number as a prime. There is a trade-off between efficiency and probability, but one can gain extremely high assurance without making unreasonable sacrifices in efficiency.
Modulus	In the context of public key cryptography, a value by which all other values are reduced. That is, if a number is bigger than the modulus, the value of the number is considered to be the same as if the number were the remainder after dividing the number by the modulus.
Near-collision resistance	Given a plaintext value and the corresponding hash value, it should be computationally unfeasible to find a second plaintext value that gives the same hash value.
NIST	The National Institute of Standards and Technology is a division of the U.S. Department of Commerce. NIST issues standards and guidelines, with the hope that they will be adopted by the computing community.

Non-repudiation	The capability of establishing that a message was signed by a particular entity. That is, a message is said to be non-repudiatable when a user sends it, and one can prove that the user sent it. In practice, cryptography can demonstrate that only particular key material was used to produce a message. There are always legal defenses such as stolen credentials or duress.
Nonce	A value used with a cryptographic algorithm that must be unique in order to maintain the security of the system. Generally, the uniqueness requirement holds only for a single key — meaning that a {key, nonce} pair should never be reused. *See also: Initialization vector, salt.*
OCB mode	*See: Offset Code Book mode.*
OCSP	*See: Online Certificate Status Protocol.*
OCSP responder	The server side software that answers OCSP requests. *See also: Online Certificate Status Protocol.*
OFB mode	*See: Output Feedback mode.*
Offset Code Book mode	A patented encryption mode for block ciphers that provides both secrecy and message integrity and is capable of doing so at high speeds.
OMAC	One-key CBC-MAC. A secure, efficient way for turning a block cipher into a message authentication code. It is an improvement of the CBC-MAC, which is not secure in the arbitrary case. Other CBC-MAC variants use multiple keys in order to fix the problem with CBC-MAC. OMAC uses a single key and still has appealing provable security properties.
One-time pad	A particular cryptographic system that is provably secure in some sense, but highly impractical, because it requires a bit of entropy for every bit of message.
One-time password	A password that is only valid once. Generally, such passwords are derived from some master secret — which is shared by an entity and an authentication server — and are calculated via a challenge-response protocol.
One-way hash function	A hash function, where it is computationally unfeasible to determine anything about the input from the output.
Online Certificate Status Protocol	A protocol for determining whether a digital certificate is valid in real time without using CRLs. This protocol (usually abbreviated OCSP) is specified in RFC 2560.
Output Feedback mode	A block cipher mode that turns a block cipher into a stream cipher. The mode works by continually encrypting the previous block of keystream. The first block of keystream is generated by encrypting an initialization vector.

Padding	Data added to a message that is not part of the message. For example, some block cipher modes require messages to be padded to a length that is evenly divisible by the block length of the cipher — i.e., the number of bytes that the cipher processes at once.
PAM	Pluggable Authentication Modules is a technology for abstracting out authentication at the host level. It is similar to SASL, but is a bit higher up in the network stack and tends to be a much easier technology to use, particularly for system administrators, who can configure authentication policies quite easily using PAM. *See also: SASL.*
Partial collision resistance	When it is unfeasible to find two arbitrary inputs to a hash function that produce similar outputs — i.e., outputs that differ in only a few bits.
Passive attack	*See: eavesdropping attack.*
Passphrase	A synonym for “password,” meant to encourage people to use longer (it is hoped, more secure) values.
Password	A value that is used for authentication.
PBKDF2	Password-Based Key Derivation Function #2. An algorithm defined in PKCS #5 for deriving a random value from a password.
PEM encoding	A simple encoding scheme for cryptographic objects that outputs printable values (by Base 64 encoding a DER-encoded representation of the cryptographic object). The scheme was first introduced in Privacy Enhanced Mail, a defunct way of providing E-mail security.
Perfect forward secrecy	Ensuring that the compromise of a secret does not divulge information that could lead to the recovery of data protected prior to the compromise. *See also: Forward secrecy.*
PKCS #1	Public Key Cryptography Standard #1. A standard from RSA Labs specifying how to use the RSA algorithm for encrypting and signing data.
PKCS #1 padding	This form of padding can encrypt messages up to 11 bytes smaller than the modulus size in bytes. You should not use this method for any purpose other than encrypting session keys or hash values.
PKCS #10	Describes a standard syntax for certification requests.
PKCS #11	Specifies a programming interface called Cryptoki for portable cryptographic devices of all kinds.
PKCS #3	Public Key Cryptography Standard #3. A standard from RSA Labs specifying how to implement the Diffie-Hellman key exchange protocol.
PKCS #5	Public Key Cryptography Standard #5. A standard from RSA Labs specifying how to derive cryptographic keys from a password.

PKCS #7	Public Key Cryptography Standard #7. A standard from RSA Labs specifying a generic syntax for data that may be encrypted or signed.
PKI	*See: Public Key Infrastructure.*
Plaintext	An unencrypted message. *See also: Ciphertext.*
PMAC	The MAC portion of the OCB block cipher mode. It is a patented way of turning a block cipher into a secure, parallelizable MAC.
Precomputation attack	Any attack that involves precomputing significant amounts of data in advance of opportunities to launch an attack. A dictionary attack is a common precomputation attack.
Private key	In a public key cryptosystem, key material that is bound tightly to an individual entity that must remain secret in order for there to be secure communication.
Privilege separation	A technique for trying to minimize the impact that a programming flaw can have, where operations requiring privilege are separated out into a small, independent component (hopefully audited with care). Generally, the component is implemented as an independent process, and it spawns off a non-privileged process to do most of the real work. The two processes keep open a communication link, speaking a simple protocol.
PRNG	*See: Pseudo-random number generator.*
Pseudo-random number generator	An algorithm that takes data and stretches it into a series of random-looking outputs. Cryptographic pseudo-random number generators may be secure if the initial data contains enough entropy. Many popular pseudo-random number generators are not secure. *See also: Stream cipher.*
Public key	In a public key cryptosystem, the key material that can be published publicly without compromising the security of the system. Generally, this material must be published; its authenticity must be determined definitively.
Public Key Infrastructure	A system that provides a means for establishing trust as to what identity is associated with a public key. Some sort of Public Key Infrastructure (PKI) is necessary to give reasonable assurance that one is communicating securely with the proper party, even if that infrastructure is ad hoc.”
RA	*See: Registration Authority.*
Race condition	A class of error in environments that are multi-threaded or otherwise multi-tasking, where an operation is falsely assumed to be atomic. That is, if two operations overlap instead of being done sequentially, there is some risk of the resulting computation not being correct. There are many cases where such a condition can be security critical.

	See also: TOCTOU problem.
Randomness	A measure of how unguessable data is. *See also: Entropy.*
RC2	A block cipher with variable key sizes and 64-bit blocks.
RC4	A widely used stream cipher that is relatively fast but with some significant problems. One practical problem is that it has a weak key setup algorithm, though this problem can be mitigated with care. Another more theoretical problem is that RC4's output is easy to distinguish from a truly random stream of numbers. This problem indicates that RC4 is probably not a good long-term choice for data security.
RC5	A block cipher that has several tunable parameters.
Registration Authority	An organization that is responsible for validating the identity of entities trying to obtain credentials in a Public Key Infrastructure. See also: *Certification Authority; Public Key Infrastructure.*
Rekeying	Changing a key in a cryptographic system.
Related key attack	A class of cryptographic attack where one takes advantage of known relationships between keys to expose information about the keys or the messages those keys are protecting.
Revocation	In the context of Public Key Infrastructure, the act of voiding a digital certificate. *See also: Public Key Infrastructure; X.509 certificate.*
RIPEMD-160	A cryptographic hash function that is well regarded. It has a 160-bit output, and is a bit slower than SHA1.
RMAC	A construction for making a Message Authentication Code out of a block cipher. It is not generally secure in the way that OMAC is, and is generally considered not worth using due to the existence of better alternatives. *See also: OMAC.*
Rollback attack	An attack where one forces communicating parties to agree on an insecure protocol version.

Root certificate	A certificate that is intrinsically trusted by entities in a Public Key Infrastructure — generally should be transported over a secure medium. Root certificates belong to a Certification Authority and are used to sign other certificates that are deemed to be valid. When a system tries to establish the validity of a certificate, one of the first things that should happen is that it should look for a chain of trust to a known, trusted root certificate. That is, if the certificate to be validated is not signed by a root, one checks the certificate(s) used to sign it to determine if those were signed by a root cert. Lather, rinse, repeat. *See also: Public Key Infrastructure.*
Round	In a block cipher, a group of operations applied as a unit that has an inverse that undoes the operation. Most block ciphers define a round operation and then apply that round operation numerous times — though often applying a different key for each round, where the round key is somehow derived from the base key.
RSA	A popular public key algorithm for encryption and digital signatures invented by Ron Rivest, Adi Shamir and Leonard Adleman. It is believed that, if factoring large numbers is computationally unfeasible, then RSA can be used securely in practice.
RSASSA-PSS	A padding standard defined in PKCS #1, used for padding data prior to RSA signing operations.
S/Key	A popular one-time password system. *See also: One-time password.*
S/MIME	A protocol for secure electronic mail standardized by the IETF. It relies on standard X.509-based Public Key Infrastructure.
SACL	System Access Control List. In Windows, the part of an ACL that determines audit logging policy. *See also: Access Control List; DACL.*
Salt	Data that can be public but is used to prevent against precomputation attacks. *See also: Initialization vector; Nonce.*
SASL	The Simple Authentication and Security Layer, which is a method for adding authentication services to network protocols somewhat generically. It is also capable of providing key exchange in many circumstances.
Secret key	*See: Symmetric key.*
Secure Socket Layer	A popular protocol for establishing secure channels over a reliable transport, utilizing a standard X.509 Public Key Infrastructure for authenticating machines. This protocol has evolved into the TLS protocol, but the term SSL is often used to generically refer to both.

	See also: Transport Layer Security.
Seed	A value used to initialize a pseudo-random number generator. *See also: Entropy, initialization vector, Pseudo-random number generator.*
Self-signed certificate	A certificate signed by the private key associated with that certificate. In an X.509 Public Key Infrastructure, all certificates need to be signed. Since root certificates have no third-party signature to establish their authenticity, they are used to sign themselves. In such a case, trust in the certificate must be established by some other means.
Serpent	A modern block cipher with 128-bit blocks and variable-sized keys. A finalist in the AES competition, Serpent has a higher security margin by design than other candidates, and is a bit slower on typical 32-bit hardware as a result. *See also: AES.*
Session key	A randomly generated key used to secure a single connection and then discarded.
SHA-256	A cryptographic hash function from NIST with 256-bit message digests.
SHA-384	SHA-512 with a truncated digest (as specified by NIST).
SHA-512	A cryptographic hash function from NIST with 512-bit message digests.
SHA1	A fairly fast, well regarded hash function with 160-bit digests that has been standardized by the National Institute of Standards and Technology (NIST).
Shared secret	A value shared by parties that may wish to communicate, where the secrecy of that value is an important component of secure communications. Typically, a shared secret is either an encryption key, a MAC key, or some value used to derive such keys.
Shatter attack	A class of attack on the Windows event system. The Windows messaging system is fundamentally fragile from a security perspective because it allows for arbitrary processes to insert control events into the message queue without sufficient mechanisms for authentication. Sometimes messages can be used to trick other applications to execute malicious code.
Single sign-on	Single sign-on allows you to access all computing resources that you should be able to reach by using a single set of authentication credentials that are presented a single time per login session. Single sign-on is a notion for improved usability of security systems that can often increase the security exposure of a system significantly.

Snooping attacks	Attacks where data is read off a network while in transit without modifying or destroying the data.
SNOW	A very fast stream cipher that is patent-free and seems to have a very high security margin.
SQL Injection	When an attacker can cause malicious SQL code to run by maliciously modifying data used to compose an SQL command.
SSL	*See: Secure Socket Layer.*
Stack smashing	Overwriting a return address on the program execution stack by exploiting a buffer overflow. Generally, the implication is that the return address gets replaced with a pointer to malicious code. *See also: Buffer overflow.*
Station-to-station protocol	A simple variant of the Diffie-Hellman key exchange protocol that provides key agreement and authenticates each party to the other. This is done by adding digital signatures (which must be done carefully). *See also: Diffie-Hellman key exchange.*
Stream cipher	A pseudo-random number generator that is believed to be cryptographically strong and always produces the same stream of output given the same initial seed (i.e., key). Encrypting with a stream cipher consists of combining the plaintext with the keystream, usually via XOR. *See also: Pseudo-random number generator.*
Strong collision resistance	Strong collision resistanceis a property that a hash function may have (and a good cryptographic hash function will have), characterized by it being computationally unfeasible to find two arbitrary inputs that yield the same output. *See also: Hash function; Weak collision resistance.*
Surreptitious forwarding	An attack on some public key cryptosystems where a malicious user decrypts a digitally signed message and then encrypts the message using someone else's public key: giving the end receiver the impression that the message was originally destined for them.
Symmetric cryptography	Cryptography that makes use of shared secrets as opposed to public keys.
Symmetric key	*See: Shared secret.*
Tag	The result of applying a keyed message authentication code to a message. *See also: Message Authentication Code.*
Tamper-proofing	*See: Anti-tampering.*

Threat model	A representation of the system threats that are expected to be reasonable. This includes denoting what kind of resources an attacker is expected to have, in addition to what kinds of things the attacker may be willing to try to do. Sometimes called an architectural security assessment.
Time of check, time of use problem	*See: TOCTOU problem.*
TLS	*See: Transport Layer Security.*
TMAC	A two-keyed variant of the CBC-MAC that overcomes the fundamental limitation of that MAC. *See also: Message Authentication Code; CBC-MAC; OMAC.*
TOCTOU problem	Time-of-check, time-of-use race condition. A type of race condition between multiple processes on a file system. Generally what happens is that a single program checks some sort of property on a file, and then in subsequent instructions tries to use the resource if the check succeeded. The problem is that — even if the use comes immediately after the check — there is often some significant chance that a second process can invalidate the check in a malicious way. For example, a privileged program might check write privileges on a valid file, and the attacker can then replace that file with a symbolic link to the system password file. *See also: Race condition.*
Transport Layer Security	The successor to SSL, a protocol for establishing secure channels over a reliable transport, using a standard X.509 Public Key Infrastructure for authenticating machines. The protocol is standardized by the IETF. *See also: Secure Socket Layer.*
Triple DES	A variant of the original Data Encryption Standard that doubles the effective security. Often abbreviated to 3DES. The security level of 3DES is still considered to be very high, but there are faster block ciphers that provide comparable levels of security — such as AES.
Trojan	*See: Backdoor.*
Trojan Horse	*See: Backdoor.*
Trusted third party	An entity in a system to whom entities must extend some implicit trust. For example, in a typical Public Key Infrastructure, the Certification Authority constitutes a trusted third party.
Twofish	A modern block cipher with 128-bit blocks and variable-sized keys. A finalist in the AES competition; it is an evolution of the Blowfish cipher. *See also: AES; Blowfish.*

UMAC	A secure MAC based on a set of universal hash functions that is extremely fast in software but so complex that there has never been a validated implementation. *See also: Universal hash function.*
Universal hash function	A keyed hash function that has ideal hash properties. In practice, the only practical functions of this nature are really &quot;almost universal&quot; hash functions, meaning they come very close to being ideal. Universal and near-universal hash functions are not cryptographically secure when used naively for message authentication but can be adapted to be secure for this purpose easily. *See also: Cryptographic hash function; Hash function; one-way hash function.*
Validation	The act of determining that data is sound. In security, generally used in the context of validating input.
Weak collision resistance	A property that a hash function may have (and a good cryptographic hash function will have), characterized by it being unfeasible to find a second input that produces the same output as a known input. *See also: Hash function; Strong collision resistance.*
Whitelist	When performing input validation, the set of items that, if matched, results in the input being accepted as valid. If there is no match to the whitelist, then the input is considered invalid. That is, a whitelist uses a &quot;default deny&quot; policy. *See also: Blacklist; Default deny.*
Window of vulnerability	The period of time in which a vulnerability can possibly be exploited.
X.509 certificate	A digital certificate that complies with the X.509 standard (produced by ANSI).
XCBC-MAC	A three-key variant of the CBC-MAC that overcomes the fundamental limitation of that MAC. *See also: Message Authentication Code; CBC-MAC, OMAC.*
XMACC	A patented parallelizable Message Authentication Code.
XSS	*See: Cross-site scripting.*