March, 31th - 2014

**Security Architecture and Models**

Written By

**Lance West**

**Table of Content**

|  |  |
| --- | --- |
| **Security Architecture and Design**  System architecture……………………….p3  The computing Industry…………………..p3  Architecture Fundamental………………...p4  Computer Architecture……………………p4  Memory Stacks……………………………p6  Multiprocessing…………………………...p7  Process management……………………...p7  Thread management………………………p8  Memory Protection Issues………………..p9  **Buffer Overflows**  Heap overflow…………………………….p9  Stack Buffer Overflow……………………p10  Some solutions for Memory Protection…..p11  **Building in Security**  Vulnerability Prevention Techniques…….p13  Table of Security Protocols Checklist……p14  Encryption………………………………...p15  Hardware Modification…………………...p16  **Operating System Architectures**  Architecture Types………………………..p18  System Security Models………………….p18  Where Do We Place Security…………….p18  **System Development Life Cycle**  Initiation…………………………………..p19  Implementation…………………………...p20  Operations/Maintenance………………….p20  Disposal…………………………………..p20  SDLC & Security Consideration Table…..p21  Threat modeling…………………………..p22  Testing/Validation Phase…………………p22  Testing Types……………………………..p23 | **The Top Risks Identified**……………….p23    **Software Development Models**  Waterfall Model………………….p24  V-Shaped Model (V-Model)……..p24  Prototyping……………………….p24  Incremental Model……………….p24  Spiral Model……………………...p24  Rapid Application Development…p24  Agile Model………………………p24  **Conclusion……………………………….**p25  **References……………………………….**p26 |

**System architecture**

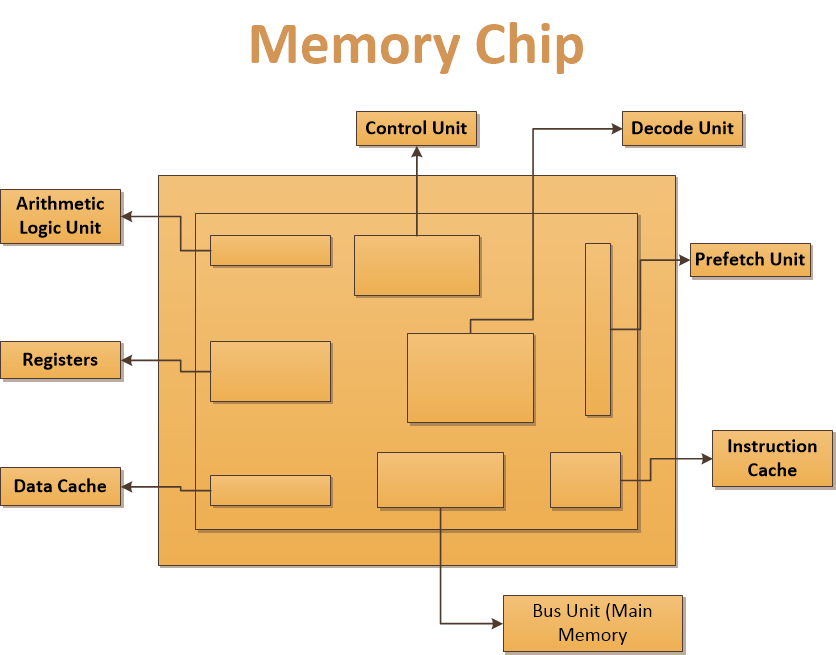
Software security is best if it is designed and built into the foundation of operating systems and applications and not added on as an afterthought. So before diving into system architecture and software security, just what does it mean? At its very basics before you start designing anything, architectural design needs to answer some key questions such as “Why are we building this system?,” “Who is going to use it and why?,” “How is it going to be used?,” “What environment will it work within?,” “What type of security and protection is required?” and “What does it need to be able to communicate with?”Any software you design will have vulnerabilities because no one has ever designed a perfect system. But if you cover the basics you will make it harder for your average hacker to gain access and exploited key components. Moreover, it will be easier to maintain and scale-up as new threats emerge within hardware and software designs.

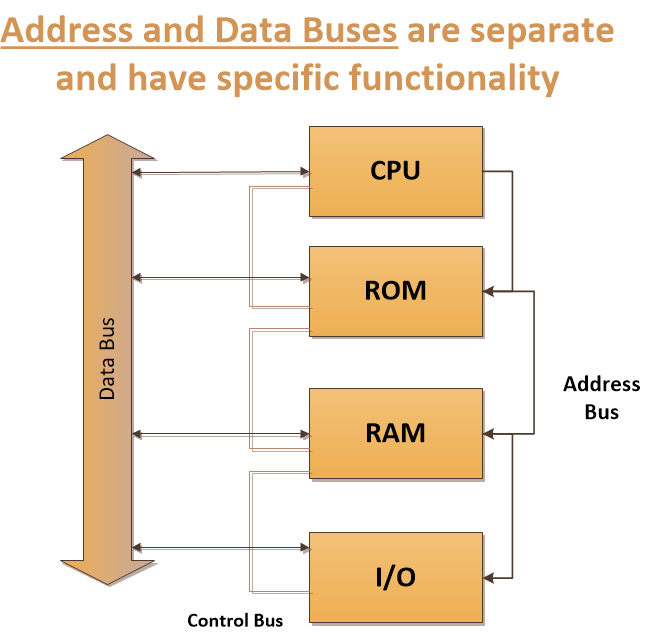
**The computing Industry** - The industry as a whole has been demanding extensive functionality, interoperability, portability at the expense of security for some time. But things are changing because governments and consumers have started questioning the high rate of identity theft. In addition, governments are now starting to take action given the amount of cyber espionage and network attacks happening across the world being done by criminal enterprises and governments

Now some industries and governments do have “trusted” systems, which provide a higher level of security than the average Windows, Linux, UNIX, or Macintosh systems. These systems have been built from the “ground up” with security as one of their main goals, so their architectures are different from consumer systems. These trusted systems are not considered general-purpose systems: they have specific functions, are expensive, are more difficult to manage, and are more commonly used by government and military environments. But given that, a lot of them have still been hacked over the last couple of years, leaves one to wonder; just how secure are these harden systems? Organizations are not just concerned about e-mail messages being encrypted as they pass through the Internet. They are also concerned about the confidential data stored in their databases, the security of their web farms that are connected directly to the Internet, and the integrity of data-entry values going into applications.

Many of these security issues must be thought through before and during the design and architectural phases for a product. Of course a company wants a piece of software to provide solid confidentiality, but does the person who is actually purchasing this software product know the correct questions to ask and what to look for? Does this person ask the vendor about cryptographic key protection, encryption algorithms, and what software development lifecycle model the vendor followed? Does the purchaser know to ask about hashing algorithms, message authentication codes, fault tolerance, and built-in redundancy options? The term standard has more than one meaning in the industry and what is a best practice? Some will follow the ISO/IEC 2700 series of standard, but these are not the only ones. IEEE came up with a standard (Standard 1471) called “Recommended Practice for Architectural Description of Software-Intensive Systems.” This was adopted by ISO and published in 2007 as ISO/IEC 42010:2007, so it’s a start.

**Architecture Fundamental -** The stakeholders for a system are the users, operators, maintainers, developers, and suppliers. Each stakeholder has his own concern pertaining to the system, which can be performance, functionality, security, maintainability, quality of service, usability, etc. The system architecture needs to express system data pertaining to each concern of each stakeholder, which is done through views. The views of the system can be logical, physical, structural, or behavioral. The creation and use of system architect processes are evolving, becoming more disciplined and standardized.

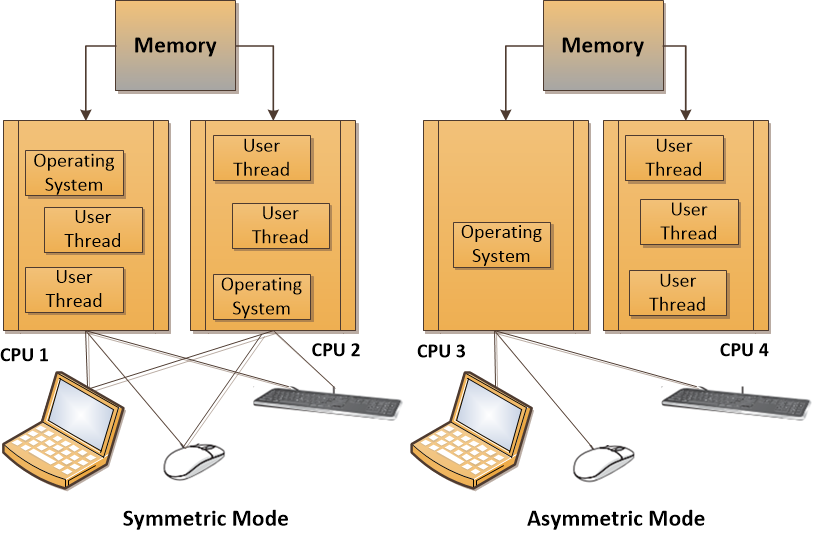
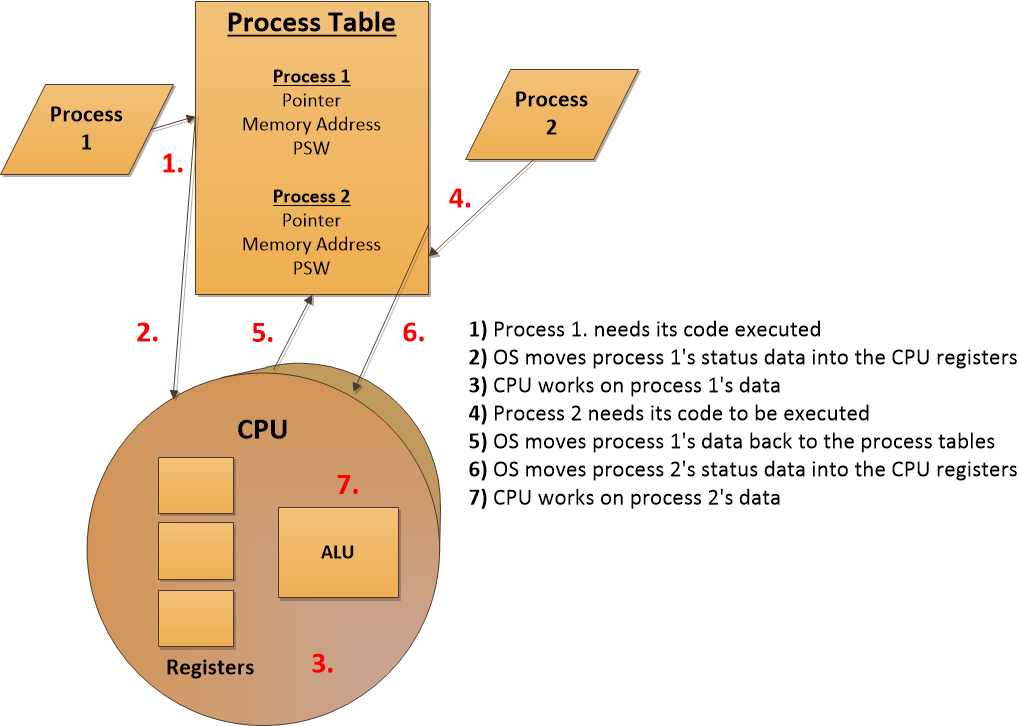
**Computer Architecture** - Computer architecture encompasses all of the parts of a computer system that are necessary for it to function, including the operating system, memory chips, logic circuits, storage devices, input and output devices, security components, buses, and networking interfaces. The interrelationships and internal working of all of these parts can be quite complex, and making them work together in a secure fashion consists of complicated methods and mechanisms. The actual execution of the instructions is done by the arithmetic logic unit (ALU). The ALU performs mathematical functions and logical operations on data. The ALU can be thought of as the brain of the CPU, and the CPU as the brain of the computer. The control unit manages and synchronizes the system while different applications’ code and operating system instructions are being executed. The control unit is the component that fetches the code, interprets the code, and oversees the execution of the different instruction sets. The program counter register contains the memory address of the next instruction to be fetched. After that instruction is executed, the program counter is updated with the memory address of the next instruction set to be processed.

Memory addresses of the instructions and data to be processed are held in registers until needed by the CPU. The CPU is connected to an address bus, which is a hardwired connection to the RAM chips in the system and the individual input/output (I/O) devices. Memory is cut up into sections that have individual addresses associated with them. I/O devices (CD-ROM, USB device, printers, and so on) are also allocated specific unique addresses. If the CPU needs to access some data, either from memory or from an I/O device, it sends a fetch request on the address bus. The fetch request contains the address of where the needed data are located. The circuitry associated with the memory or I/O device recognizes the address the CPU sent down the address bus and instructs the memory or device to read the requested data and put it on the data bus.

This process is illustrated in Figure 4-4.Once the CPU is done with its computation, it needs to return the results to the requesting program’s memory. So, the CPU sends the requesting program’s address down the address bus and sends the new results down the data bus with the command write. These new data are then written to the requesting program’s memory space. The address and data buses can be 8, 16, 32, or 64 bits wide. Most systems today use a 64-bit address bus, which means the system can have a large address space (2^64). Systems can also have a 64-bit data bus, which means the system can move data in parallel back and forth between memory, I/O devices, and the CPU of this size.

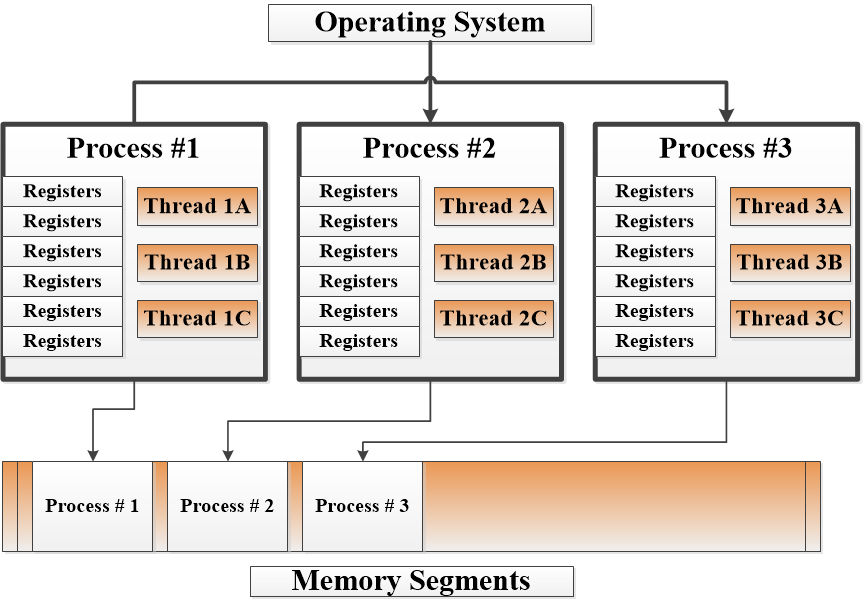
**Memory Stacks** - Each process has its own stack, which is a data structure in memory that the process can read from and write to in a last in, first out (LIFO) fashion. Let’s say you and I need to communicate through a stack. What I do is put all of the things I need to say to you in a stack of papers. The first paper tells you how you can respond to me when you need to, which is called a return pointer. The next paper has some instructions I need you to carry out. The next piece of paper has the data you must use when carrying out these instructions. So, I write down on individual pieces of paper all that I need you to do for me and stack them up. When I am done, I tell you to read my stack of papers. You take the first page off the stack and carry out the request. Then you take the second page and carry out that request. You continue to do this until you are at the bottom of the stack, which contains my return pointer.

You look at this return pointer (which is my memory address) to know where to send the results of all the instructions I asked you to carry out. This is how processes communicate to other processes and to the CPU. One process stacks up its information that it needs to communicate to the CPU. The CPU has to keep track of where it is in the stack, which is the purpose of the stack pointer. Once the first item on the stack is executed, then the stack pointer moves down to tell the CPU where the next piece of data is located.

**Multiprocessing** **-** If the computer system is configured to work in symmetric mode, this means the processors are handed work as needed, as shown with CPU 1 and CPU 2 in Figure below. It is like a load-balancing environment. When a process needs instructions to be executed, a scheduler determines which processor is ready for more work and sends it on. Many resources state that today’s operating systems provide multiprogramming and multitasking. This is true, in that multiprogramming just means more than one application can be loaded into memory at the same time. But in reality, multiprogramming was replaced by multitasking, which means more than one application can be in memory at the same time and the operating system can deal with requests from these different applications simultaneously. Multiprogramming is a legacy term.

**Process management -** Cooperative multitasking, used in Windows 3.x and early Macintosh systems, required the processes to voluntarily release resources they were using. This was not necessarily a stable environment, because if a programmer did not write his code properly to release a resource when his application was done using it, the resource would be committed indefinitely to his application and thus be unavailable to other processes. With preemptive multitasking, used in Windows 9x and later versions and in Unix systems, the operating system controls how long a process can use a resource. The system can suspend a process that is using the CPU and allow another process access to it through the use of time sharing. So, in operating systems that used cooperative multitasking, the processes had too much control over resource release, and when an application hung, it usually affected all the other applications and sometimes the operating system itself. Different operating system types work within different process models. For example, Unix and Linux systems allow their processes to create new children processes, which is referred to as forking. A process can be in a running state (CPU is executing its instructions and data), ready state (waiting to send instructions to the CPU), or blocked state (waiting for input data, such as keystrokes, from a user). When a process is blocked, it is waiting for some type of data to be sent to it.

**NOTE:** Not all operating systems create and work in the process hierarchy like Unix and Linux systems. Windows systems do not fork new children processes, but instead create new threads that work within the same context of the parent process. The operating system keeps a process table, which has one entry per process. The table contains each individual process’s state, stack pointer, memory allocation, program counter, and status of open files in use. The reason the operating system documents all of this status information is that the CPU needs all of it loaded into its registers when it needs to interact with, for example, process 1. When process 1’s CPU time slice is over, all of the current status information on process 1 is stored in the process table so that when its time slice is open again, all of this status information can be put back into the CPU registers. So, when it is process 2’s time with the CPU, its status information is transferred from the process table to the CPU registers, and transferred back again when the time slice is over. These steps are shown in the Process Table above.

**Thread management** - A process is a program in memory. More precisely, a process is the program’s instructions and all the resources assigned to the process by the operating system. It is just easier to group all of these instructions and resources together and control them as one entity, which is a process. A program that has been developed to carry out several different tasks at one time (display, print, interact with other programs) is capable of running several different threads simultaneously. An application with this capability is referred to as a multithreaded application. As the complexity of our systems increases, the potential of truly securing them decreases. There is an inverse relationship between complexity and security:

**Memory Protection Issues -** There is many different types of memory and each reacts and works different within a computer system. All of these issues make it more difficult for memory management to be carried out properly in a constantly changing and complex system. Here is just some example of why memory protection is so difficult.

* Every address reference may need to be validated for protection.
* Two or more processes can share access to the same segment with potentially different access rights.
* Different instruction and data types can be assigned different levels of protection.
* Processes cannot generate an unpermitted address or gain access to an unpermitted segment.

Moreover, some of the most effective attacks that have happen on computer systems relates to memory. Target Stores are a good example is one of the most recent attacks that was done using what is a called a "memory scraping." hacking the data while the data is still in the memory buffer unencrypted.

**Buffer Overflows**

As we can see from the Target Store attack, memory still plays a big part in computer viruses, and computer vulnerabilities. That is why I have chosen to concentrate on Buffer Overflow attacks given their history. A buffer overflow takes place when too much data are accepted as input to a specific process within an allocated segment of memory. A buffer can be overflowed arbitrarily with too much data, but for it to be of any use to an attacker, the code inserted into the buffer must be of a specific length, followed up by commands the attacker wants executed. So, the purpose of a buffer overflow may be either to make a mess or insert scripts to take control of the system. There are two main types of memory buffer overflows, the stack overflow and the heap overflow.

**Heap overflow -** contains dynamically assigned variables. One method is to exploit dynamic memory allocation to cause created pointer to overwrite function pointers. This allows the attacker to have custom functions run instead of the ones originally programmed. Below is an simple example of a “Heap Buffer Overflow,” attack.

#define BUFSIZE 16

#define OVERSIZE 8 /\*overflow buf2 by Oversize bytes \*/

int main()

{

u\_long diff;

char \*buf1 (char\*)malloc(BUFSIZE), \*buf2 = (char\*)malloc(BUFSIZE)

diff = (u\_long)buf2 - (u\_long)buf1;

printf("buf1 = %p, buf2 = %p, diff = 0x%x bytes\n", buf1, buf2, diff)

memset(buf2, 'A', BUFSIZE-1), buf2[BUFSIZE\*1] = '\0';

printf("before overflow: buf2 = %s\n", buf2);

memset(buf1, 'B', (u\_int)(diff + OVERSIZE));

print('after overflow: buf2 = %s\n", buf2);

return 0;

}

**Output from heap buffer Overflow.**

buf1=0x804e0000, buf2 = ox8044eff0, diff = 0xff0 bytes

before overflow: buf2 = AAAAAAAAAAAAAAA

after overflow: buf2 = BBBBBAAAAAA

Buf1 overruns its boundaries into buf2's heap space, But because buf2's heap space is still valid heap memory, the program doesn't crash

**Stack Buffer Overflow -** A stack buffer overflow occurs when a program writes more data to a buffer located on the stack than there was actually allocated for that buffer. This almost always results in corruption of adjacent data on the stack, and in cases where the overflow was triggered by mistake, will often cause the program to crash or operate incorrectly. If the affected program is running with special privileges, or accepts data from untrusted network hosted webserver then the bug is potential security vulnerability. If the stack buffer is filled with data supplied from an untrusted user then that user can corrupt the stack in such a way as to inject executable code into the running program and take control of the process. This is one of the oldest and more reliable methods for attackers. Below is a simple example of a stack overflow attack.

#include <string.h>

void foo (char \*bar)

{

char c[12];

strcpy(c, bar); //no bounds checking

}

int main (int argc, char \*\*argv)

{

foo(argv[1]);

}

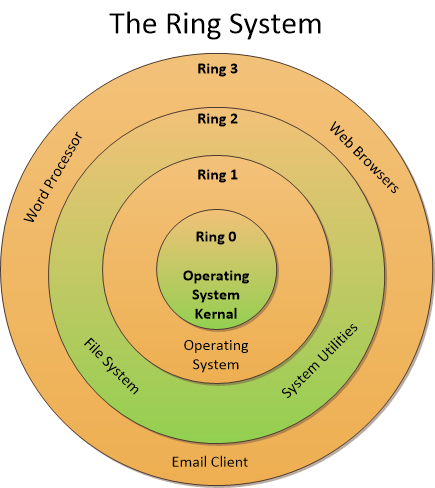
If for example your input is more than 11 bytes then the local data stack, frame pointer, and return address will be overwritten making it possible to redirect the function to the attacker’s custom functions. It could even allow the code elevated privileges to gain access to the root of the system.

**Some Solutions for Memory Protection:**

* Randomize the stack memory allocation
* Use Canaries (word guards) to detect buffer overflows
* Stay away from 'libc' string calls in C++
* Use Stack-Guard, Pro-Police to automate the protection of stacks
* Check your bounds within your C++ code. Functions that do perform the necessary boundary checking include strncpy(), strncat(), snprintf(), and vsnprintf().

Vendors of different operating systems (Windows, Unix, Linux, Macintosh, etc.) have implemented various types of protection methods integrated into their memory manager processes. For example, Windows Vista was the first version of Windows to implement address space layout randomization (ASLR), which was first implemented in OpenBSD. If an attacker wants to maliciously interact with a process, he needs to know what memory address to send his attack inputs to. If the operating system changed these addresses continuously, which is what ALSR accomplishes this would greatly reduce the potential success of his attack. You can’t mess with something if you don’t know where it is.

Many of the main operating systems use some form of data execution prevention (DEP), which can be implemented via hardware (CPU) or software (operating system). The actual implementations of DEP varies, but the main goal is to help ensure that executable code does not function within memory segments that could be dangerous. It is similar to not allowing someone suspicious in your house. You don’t know if this person is really going to do something malicious, but just to make sure you will not allow him to be in a position where he could bring harm to you or your household. DEP can mark certain memory locations as “off limits” with the goal of reducing the “playing field” for hackers and malware.

The operating system has several other protection mechanisms to ensure processes do not negatively affect each other or the critical components of the system itself. One has already been mentioned: memory protection. Another security mechanism the system uses is a ring-based architecture. The architecture of the CPU dictates how many rings are available for an operating system to use. The ring system diagram: Within the ring system the most protected code is stored in the center of the facility (ring 0), where only the most trusted code is allowed to run. The next most trusted in (ring 1) where the operating systems lives, and the outer rings like (Ring 2) is used for system utilities and (Ring 3) the most outer ring is for applications and web browsers. So when you build an application you can put in rules that if the application wants to access another level within you ring system. That program will have to meet or excess protocols in order to gain access. Like stopping them from inheriting objects and memory or giving them rights to run executable code within the inter rings.

**Building in Security**

Here is just a quick list of some of the what, when, and why something’s are happening and something’s are not within the cyber security world. Why there is a growing awareness of common security issues, and why organization continues to face cyber challenges:

* Security may not be a priority in the customer purchase decision.
* The organization may lack a formal security policy and specific security requirements.
* The product development team may lack security awareness and knowledge.
* The organization may lack uniform or widely applicable security considerations, practices, tools, and techniques for the development of secure products.
* The development team may place primary focus on functionality, and security becomes secondary and often an afterthought.
* Security activities in the schedule/project plan may encounter resource constraints.
* Perceived cost and ongoing maintenance of security may not be viewed as affordable.
* Despite the wide- scale awareness of common security flaws in software products, e.g., buffer overflows, resource exhaustion, and structured query language (SQL) injection, the same flaws continue to exist in some of the current products

**Vulnerability Prevention Techniques -** One of the best ways to prevent the exploitation of buffer overflow vulnerabilities is to detect and eliminate them from the source code before the software is put to use, usually by performing some sort of static analysis on either the source code or on the compiled binaries. A proven technique for uncovering flaws in software is source code review, also known as source code auditing. Tools designed for automatic source code analysis complement manual audits by identifying potential security violations, including functions that perform unbounded string copying. Some of the best-known tools are ITS4 (www.cigital.com/its4/), RATS (www.securesw.com/rats/), and LCLint [7]. An extensive list of auditing tools is provided by the Sardonix portal at sardonix.org/Auditing\_Resources.html.

If the source code is available, a developer can add buffer overflow detection automatically to a program by using a modified compiler; four such compilers are Stack-Guard, Pro-Police, Stack-Shield, and Return Address Defender (RAD). Because many buffer overflow attacks take place by loading executable code onto the stack and redirecting execution there, one of the simpler approaches to defending against them is to modify the stack segment so it is non-executable. Most Unix-like operating systems have an optional patch or configuration switch that removes execute permissions from the program stack. Many security-critical applications are compiled statically, making it possible in some instances for a determined attacker to bypass the modified libraries.

**Table of Security Protocols Checklist**

|  |  |  |
| --- | --- | --- |
| **Phase** | **Key security considerations** | **Example checklists** |
| **Requirements** | •Determination of security assumptions, including product deployment scenarios.  •Identification of critical assets to be protected and secured  •Identification of security requirements and interface specifications for third-part product incorporated  in the design and security interoperating with rest of the network  •Identification of requirements for securing the communication, data storage, and configuration for the product.  •Performing high-level threat analysis  •Determination of the product hardening techniques to be applied. | •Regulatory list for industry.  •List of relevant best practices to be incorporated.  •List of standards for compliance.  •List of high-level threats. |
| **Design** | •Define and design security architectures where product will reside  •Perform detailed threat and potential vulnerability analysis for critical assets.  •Principles of defense in depth; least privilege and partitioning should be followed.  •Both static and dynamic analysis tools should be used. | •Access Controls, Authentication, Confidentiality  •List of advanced secure protocol standards  •completion of applicable architecture views for management signaling, and user. |
| **Implementation** | •Ensuring a secure development environment.  •Adhering to security standards and best practices for protocol implementation, hardening, and coding practices.  •Use of secure tools, e.g., compilers, implementation reviews. Code reviews are the primary mechanisms for ensuring the security in the implementation phase. | •Configuration guidelines.  •List of unused ports to be protected.  •List of rules for product hardening to guide implementation choices. |
| **Testing** | •Determine whether security mechanisms are working as designed and whether anything is missing.  •Determine whether software implementation has introduced new vulnerabilities that can be exploited.  •Defect review analysis is helpful in preventing further security defects in the development cycles.  •Apply stress testing for vulnerabilities and penetration. | •List of adversarial test cases to be executed.  •List of known vulnerabilities in subcomponents ad in the product that have been tested.  •Use of secure static and dynamic tools, e.g., compilers, standards compliance, implementation reviews.  •Check for open ports. |

**Encryption –** Since encryption is a method of transforming readable data, called plaintext, into a form that appears to be random and unreadable, which is called ciphertext. The method you chose to use within your application will have to be based on many factors with the one main goal. If someone intercepts your data they will not have an easy time decrypting it. I will just give a quick overview of some of the key features to consider when picking an encryption protocols.

**Symmetric vs. Asymmetric Algorithms**

Symmetric Strengths/weaknesses

* Much faster (less computationally intensive) than asymmetric systems.
* Hard to break if using a large key size.
* Weaknesses
* Requires a secure mechanism to deliver keys properly.
* Each pair of users needs a unique key, so as the number of individual’s increases, so does the number of keys, possibly making key management overwhelming.
* Provides confidentiality but not authenticity or non-repudiation

Asymmetric Strengths/weaknesses

Strengths

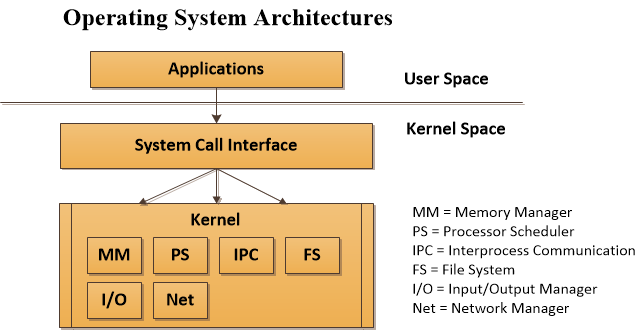
* Better key distribution than symmetric systems.
* Better scalability than symmetric systems
* Can provide authentication and non-repudiation

Weaknesses

* Works much more slowly than symmetric systems
* Mathematically intensive tasks
* The following are examples of asymmetric key algorithms:
* Rivest-Shamir-Adleman (RSA)
* Elliptic curve cryptosystem (ECC)
* Diffie-Hellman
* El Gamal
* Digital Signature Algorithm (DSA)
* Merkle-Hellman Knapsack

**Block and Stream Ciphers -** Block cipher Symmetric algorithm type that encrypts chunks (blocks) of data at a time. Stream cipher Algorithm type that generates a key-stream (random values), which is XORd with plaintext for encryption purposes.

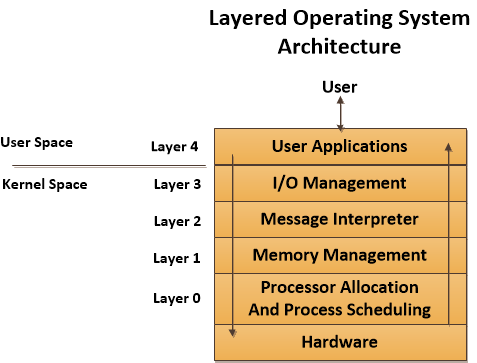
From your threat analysis you will have an idea of what needs to be encrypted. Be it a financial transactions, or database tables. Once you understand the threat it will be easier to pick the right type of encryption to match the job. Because there are many different types of encryption methodologies and each have their strength and weaknesses - key length to the amount of hardware resources each requires.

**Hardware Modification -** Any technique or program you use to detect buffer overflow attacks will exact a performance cost from the system employing it. One system called Smash-Guard uses a modification of the micro-coded instructions for the CALL and RET-opcodes in a CPU to enable transparent protection against buffer overflow attacks. Smash-Guard takes advantage of the fact that a modern CPU has substantial memory space on the chip and creates a secondary stack that holds return addresses similar to the return address repository. Unlike other programs such as Stack-Shield which pretty much does the same thing. The Smash-Guard modifications to the CPU microcode make it possible to add protection without having to modify the software - [www.smashguard.org/](http://www.smashguard.org/).

**Operating System Architectures**

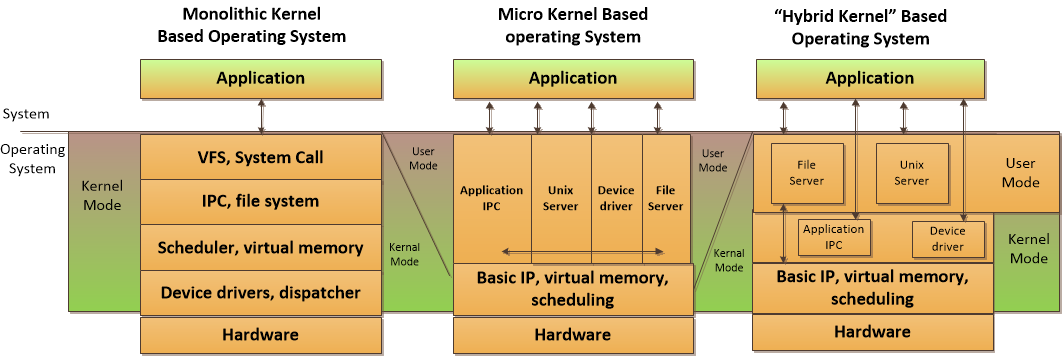
Operating system architecture which deals specifically with the software components of a system While operating systems are very complex, some main differences in the architectural approaches have come down to what is running in kernel mode and what is not. In a monolithic architecture, all of the operating system processes work in kernel mode, as illustrated in “Monolithic operating system architecture Diagram.” The services provided by the operating system (memory management, I/O management, process scheduling, file management, etc.) are available to applications through system calls.

Earlier operating systems, such as MS-DOS, were based upon a monolithic design. The whole operating system acted as one software layer between the user applications and the hardware level. There are several problems with this approach: complexity, portability, extensibility, and security. Since the functionality of the code is spread throughout the system, it is hard to test and debug. If there is a flaw in a software component it is difficult to localize and easily fix. Many pieces of code had to be modified just to address one issue.

The layered operating system architecture separates system functionality into hierarchical layers. For example, a system that followed a layered architecture was, strangely enough, called The (Technische Hogeschool Eindhoven) multiprogramming system. The had five layers of functionality. Layer 0 controlled access to the processor and provided multiprogramming functionality; layer 1 carried out memory management; layer 2 provided inter-process communication; layer 3 dealt with I/O devices; and layer 4 was where the applications resided. The processes at the different layers each had interfaces to be used by processes in layers below and above them. This layered approach, illustrated below, had the full operating system still working in kernel mode (ring 0). The main difference between the monolithic approach and this layered approach is that the functionality within the operating system was laid out in distinctive layers that called upon each other.

**Architecture Types -** The basic core definitions of the different architecture types are as follows:

* Monolithic all operating system processes run in kernel mode.
* Layered all operating system processes run in a hierarchical model in kernel mode.
* Microkernel Core operating system processes run in kernel mode and the remaining ones run in user mode.
* Hybrid microkernel all operating system processes run in kernel mode. Core processes run within a microkernel and others run in a client\server model.
* The main architectures that are used in systems today are illustrated below



**System Security Models -** An important concept in the design and analysis of secure systems is the security model, because it incorporates the security policy that should be enforced in the system. A model is a symbolic representation of a policy. It maps the desires of the policymakers into a set of rules that a computer system must follow.

Software controls come in various flavors with many different goals. They can control input, encryption, logic processing, number-crunching methods, inter-process communication, access, output, and interfacing with other software. They should be developed with potential risks in mind, and many types of threat models and risk analyses should be invoked at different stages of development.

**Where do we place security -** Today, many security efforts look to solve security problems through controls such as firewalls, intrusion detection systems (IDSs), content filtering, antivirus software, vulnerability scanners, and much more. This reliance on a long laundry list of controls occurs mainly because our software contains much vulnerability. In the past, it was not crucial to implement security during the software development stages; thus, many programmers today do not practice these procedures. Most security professionals are not software developers, and thus do not have complete insight to software vulnerability issues:

* Many software developers do not have security as a main focus. Functionality is usually considered more important than security.
* Software vendors are trying to get their products to market in the quickest possible time, and thus do not take time for proper security architecture, design, and testing steps.
* The computing community has gotten used to receiving software with flaws and then applying patches. This has become a common and seemingly acceptable practice.
* Customers cannot control the flaws in the software they purchase, so they must depend upon perimeter protection.

Security has been mainly provided by security products and perimeter devices rather than controls built into applications. The security products can cover a wide range of applications, can be controlled by a centralized management console, and are further away from application control.

**System Development Life Cycle**

I will not spend much time on the (SDLC). It’s a topic well covered in a lot of technology books. I will just give a quick overview for reference.

* Initiation Need for a new system is defined
* Acquisition/development New system is either created or purchased
* Implementation new system is installed into production environment
* Operation/maintenance System is used and cared for
* Disposal System is removed from production environment

**Initiation-** In the initiation phase the company establishes the need for a specific system. The company has figured out that there is a problem that can be solved or a function that can be carried out through some type of technology. Acquisition or Development; before the system is actually developed or purchased, several things should take place to ensure the end result meets the company’s true needs. Just a few of the activities one can follow:

* Requirements analysis In-depth study of what functions the company needs the desired system to carry out.
* Formal risk assessment Identifies vulnerabilities and threats in the proposed system and the potential risk levels as they pertain to confidentiality, integrity, and availability. This builds upon the initial risk assessment carried out in the previous phase. The results of this assessment help the team build the system’s security plan.
* Security functional requirements analysis identifies the protection levels that must be provided by the system to meet all regulatory, legal, and policy compliance needs.
* Security assurance requirements analysis identifies the assurance levels the system must provide. The activities that need to be carried out to ensure the desired level of confidence in the system are determined, which are usually specific types of tests and evaluations.
* Third-party evaluations reviewing the level of service and quality a specific vendor will provide if the system is to be purchased.
* Security plan documented security controls the system must contain to ensure compliance with the company’s security needs. This plan provides a complete description of the system and ties them to key company documents, as in configuration management, test and evaluation plans, system interconnection agreements, security accreditations, etc.
* Security test and evaluation plan Outlines how security controls should be evaluated before the system is approved and deployed.

**Implementation -** It may be necessary to carry out certification and accreditation (C&A) processes before a system can be formally installed within the production environment. Certification is the technical testing of a system. Established verification procedures are followed to ensure the effectiveness of the system and its security controls. Accreditation is the formal authorization given by management to allow a system to operate in a specific environment. The accreditation decision is based upon the results of the certification process.

**Operations/Maintenance -** A system should have baselines set pertaining to the system’s hardware, software, and firmware configuration during the implementation phase. In the operation and maintenance phase, continuous monitoring needs to take place to ensure that these baselines are always met.

**Disposal -** When a system no longer provides a needed function, plans for how the system and its data will make a transition should be developed. Data may need to be moved to a different system, archived, discarded, or destroyed. If proper steps are not taken during the disposal phase, unauthorized access to sensitive assets can take place.

**System Development Life Cycle (SDLC) & Security Considerations (SC) Table**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **Initiation** | **Acquisition/Development** | **Implementation** | **Operations/Maintenance** | **Disposal** |
| **SDLC** | **Needs Determination**  -Perception of a need  -Linkage of need to mission and performance object  -Assessment of alternatives to capital assets  -Preparing for investment review and budgeting | **Functional statement of need**  -Market Research  -Feasibility study  -Requirements analysis  -Alternatives analysis  -Cost-Benefit Analysis  -Software conversion study  -Risk Management Plan  -Acquisition planning  Risk assessment | Installation  Inspection  Acceptance testing  Initial User Training  Documentation | Performance Measurement  Contract Modifications  Operations  Maintenance | Appropriateness of disposal  Exchange and sale  Internal organization screening  Transfer and donation  Contract closeout |
| **Security Consideration** | -Security Categorization  -Preliminary risk assessment | -Risk assessment  -Security Functional Requirements  -Security Assurance -Requirements  Cost/Reporting  -Security Planning  -Security Control  -Developmental security test and evaluation | -Inspection and Acceptance  -System Integration  Security Certification  Security Accreditation | Configuration Management  and control  Continuous  Monitoring | Information Preservation  Media Sanitization  Hardware and Software Disposal |
|  |  |  |  |  |  |

There have been several software development life cycle (SDLC) models developed over the years but it’s only been recent that cyber security has take center stage given all the publicity from the media. It’s finally dawning on people what identity theft is, and just how damaging it can be to people in many ways. So as you built your new application always keep a few things on your mind and on the table.

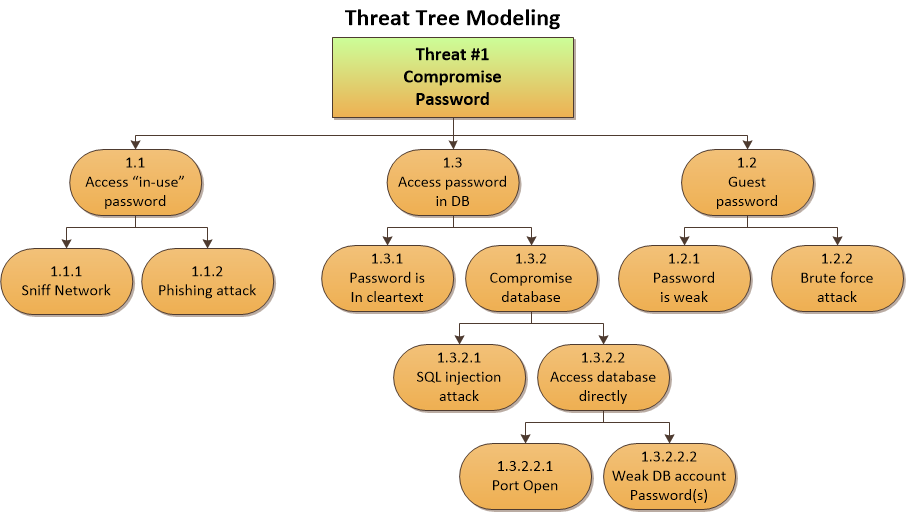
• Security requirements

• Security risk assessment

• Privacy risk assessment

• Risk-level acceptance

**Threat modeling -** A systematic approach used to understand how different threats could be realized and how a successful compromise could take place. If you were responsible for ensuring that the government building that you worked was safe from terrorist attacks, you would run through scenarios that terrorists would most likely carry out so that you fully understood how to protect the facility and the people within it. You could think through how someone could bring a bomb into the building, and then you would better understand the screening activities that need to take place at each entry point.



**Testing/Validation Phase -** It’s not easy coming up with a standard template for security testing because the applications and products can be so diverse in functionality and security objectives. But its still important to map security risks to test cases and code. Linear thinking can be followed by identifying and providing the necessary test scenario, performing the test, and reviewing the code for how it deals with different vulnerability. At this phase, tests are conducted in an environment that should mirror the production environment to ensure the code does not work only in the labs. Security attacks and penetration tests usually take place during this phase to identify any missed vulnerabilities. Functionality, performance, and penetration resistance are evaluated. All the necessary functionality required of the product should be in a checklist to ensure each function is accounted for.

**Testing Types -** If we would like the assurance that the software is any good at all, we should probably test it. There are different types of tests the software should go through because there are different potential flaws the team should be looking for. The following are some of the most common testing approaches:

* Unit testing Individual component is in a controlled environment where programmers validate data structure, logic, and boundary conditions.
* Integration testing Verifying that components work together as outlined in design specifications.
* Acceptance testing ensuring that the code meets customer requirements.
* Regression testing after a change to a system takes place, retesting to ensure functionality, performance, and protection.

Another testing technic is called: fuzzing. Fuzzing is a technique used to discover flaws and vulnerabilities in software. Fuzzing is the act of sending random data to the target program in order to trigger failures. This can be ran during dynamic analysis refers to the evaluation of a program in real time, i.e., when it is running. Dynamic analysis is carried out once a program has cleared the static analysis stage and basic programming flaws have been rectified offline. Dynamic analysis enables developers to trace subtle logical errors in the software that are likely to cause security mayhem later on. The primary advantage of this technique is that it eliminates the need to create artificial error-inducing scenarios. Dynamic analysis is also effective for compatibility testing, detecting memory leakages, and identifying dependencies, and for analyzing software without having to access the software’s actual source code.

**The top risks identified**

The top risks relating to software development writing are as follows: I am only covering one of the most frequent and still a major problems in cyber security when it comes to designing safe and secure software, which is Buffer Overflows, but there are others.

1. Buffer Overflows
2. SQL Injection
3. Cross-Site scripting (XSS)
4. Broken Authentication and Session management
5. Insecure Direct Object References
6. Cross-Site Request Forgery (CSRF)
7. Security Mis-configuration
8. Insecure Cryptographic Storage
9. Failure to Restrict URL Access
10. Insufficient Transport Layer Protection
11. Un-validated Redirects and Forwards

Now one of the keys to developing better software design is to pick a life cycle that fits your needs with a model of development that matches your culture, because without any standards you will have no way of measuring your success or failure. Everything starts someplace, and you always have the option to change and evolve as time goes on**.**

**Software Development Models**

I will not spend much time on the basic development models since this for the most part is a well travel road. I will just outline some of the more obscure ones.

**Build and Fix Model** - Basically, no architecture design is carried out in the Build and Fix model; instead, development takes place immediately.

**Waterfall Model – A** sequential approach that requires each phase to complete before the next one can begin - Difficult to integrate changes - Inflexible model.

**V-Shaped Model (V-Model)** - emphasizes verification and validation at each phase and testing to take place throughout the project, not just at the end.

**Prototyping** - creates a sample or model of the code for proof-of-concept purposes.

**Incremental Model** - multiple development cycles are carried out on a piece of

software throughout its development stages. Each phase provides a usable

version of software.

**Spiral Model** – is an iterative approach that emphasizes risk analysis per iteration. Allows for customer feedback to be integrated through a flexible evolutionary approach.

**Rapid Application Development** - combines prototyping and iterative development procedures with the goal of accelerating the software development process.

**Agile Model** – Is an iterative and incremental development processes that encourage team based collaboration - flexibility and adaptability play key roles.

**Conclusion**

A software product that has security built into it from concept to requirements, design, architecture, implementation, testing, deployment, and going live. That product will not only be harder to hack, but will be easier to scale and change as new viruses/vulnerabilities come onto the scene. It’s important that security be a critical part of any software design project, because it will not only help uncover vulnerabilities within the product at each stage of development, but it will enlighten developers and network people alike to the methodology of how to built a secure system before its deployed onto the network. It’s a beginning to a framework of knowledge that can be applied to many IT situations where security is needed, because robust security architecture does not happen by accident. It requires planning and a deep skill-set of knowledge from software, networks, and hardware. The more people understand all the moving parts, the easier it will be for them to protect it.

**References**

*Harris, Shon. (2013). All-In-One CISSP Exam Guide Sixth Edition*

*Publisher: McGraw Hill*

Kuperman, Benjamin A. (Nov 2005). DETECTION AND PREVENTION OF STACK BUFFER OVERFLOW ATTACKS. Vol. 48 Issue 11, p51-56.

Retrieved from: <http://web.b.ebscohost.com.ezproxy.metrostate.edu/ehost/pdfviewer/pdfviewer?sid=9286cc38-da42-48bb-acc4-4764523ee7e5%40sessionmgr114&vid=1&hid=114>

Daping Wang (2007). An XML-based testing strategy for probing security vulnerabilities in the diameter protocol. Vol. 12 Issue 3, p79-93.

Retrieved from: <http://web.b.ebscohost.com.ezproxy.metrostate.edu/ehost/pdfviewer/pdfviewer?sid=537a9842-4dad-45d4-a1b5-5f7d28cc4dfb%40sessionmgr111&vid=2&hid=114>

Gupta, Ashok K., Chandrashekhar, Uma(2007). Building secure products and solutions. Vol. 12 Issue 3, p21-38 - Retrieved from:

<http://web.b.ebscohost.com.ezproxy.metrostate.edu/ehost/pdfviewer/pdfviewer?sid=fbf70275-5684-4dad-9888-c08e9abfcb77%40sessionmgr111&vid=1&hid=114>

Neumann, Peter G. (2008). REFLECTIONS ON COMPUTER-RELATED RISKS. Vol. 51 Issue 1, p78-80. Retrieved from:

<http://web.b.ebscohost.com.ezproxy.metrostate.edu/ehost/pdfviewer/pdfviewer?sid=8c6d763e-a923-4cae-be69-cbecb85353d0%40sessionmgr114&vid=1&hid=114>

Fisher, Dennis (2001). Code Red Virus. Retrieved from:

<http://web.b.ebscohost.com.ezproxy.metrostate.edu/ehost/pdfviewer/pdfviewer?sid=43aeb36e-0731-4b19-8c6f-75c204daea41%40sessionmgr113&vid=1&hid=114>

Harbaugh, Logan G. (2004). DPI 100 security appliance from Barbedwire Technologies.

Retrieved from:

<http://web.b.ebscohost.com.ezproxy.metrostate.edu/ehost/pdfviewer/pdfviewer?sid=f2c6a3ab-8962-403d-8c0b-4892ea0628d1%40sessionmgr198&vid=1&hid=114>