CHAPTER 1

INTRODUCTION

1.1 DATA & INFORMATION

Information is the collection of meaningful data stored in a computer in binary form. In today's world of Big Data, the information stored in Data centres needs to be secured.

Data can be defined as a representation of facts, concepts, or instructions in a formalized manner, which should be suitable for communication, interpretation, or processing, by human or electronic machine.

Data is represented with the help of characters such as alphabets (A-Z, a-z), digits (0-9) or special characters (+,-,/,*,<,>,= etc.)

Information is organized or classified data, which has some meaningful values for the receiver. Information is the processed data on which decisions and actions are based.

For the decision to be meaningful, the processed data must qualify for the following characteristics −

- **Timely** − Information should be available when required.
- **Accuracy** − Information should be accurate.
- **Completeness** − Information should be complete.

Figure 1.1: Information Systems

SAARTHAK MAHAJAN
04496403114

You can store it, retrieve it, withhold it, protect it and it is sometimes worth a lot of money. Information has become the cornerstone that connects the world. It is everywhere -- from the largest mainframe computer to the tiny Android or iPhone in the palm of your hand. Information transforms into computer information when it is stored in a system to support calculated decision-making. There are certain characteristics computer information must possess to be useful and meaningful, work for the consumers of that system and is critical to effective system design. Computer information must be designed to be relevant, complete, timely, accurate, accessible, understandable and valuable.

Relevant

Information, to be considered relevant, must meet the requirements of the information consumer group. This means the system contains data the consumer can use or is of value in some significant way. Relevant information also contributes to the overall success of a system. Irrelevant information could result in system mortality because consumers will not use the system if the information is not necessary or has no value. System designers can avoid this fate by evaluating computer information for factors of relevancy such as importance and alignment with business or project objectives.

Complete

Information must be complete to provide consumers with a full and operational picture. Similar to a blind spot when driving a vehicle, incomplete information is a serious problem that can render a system unusable and significantly impair a user's or business' ability to make effective choices or draw accurate conclusions. Information that is not complete is as bad, if not worse, than having no information at all, because it represents a misleading or skewed view of the data. A computer information system should include a measure for managing, handling or preventing missing data.

Timely

The need for speed and timely access to computer information is paramount in this digital age, especially for businesses. To be timely, information must be dynamic and available when needed. Timely computer information can give an organization an advantage over the competition, which can positively affect the bottom line or be financially profitable. Alternatively, lack of timely information can cause costly delayed reactions. An effective way of ensuring your computer information is timely is by employing data modeling techniques or identifying patterns in your data to enable predictability.

Accessible

For information to be meaningful to consumers, it must be accessible and within reach. If information consumers are not able to access the data when they need it, it can lead to frustration. An example of an issue that can cause computer information to be inaccessible is performance. If a system is slow or goes down periodically, it can affect a consumer's ability to perform job duties effectively. Organizations can benefit

SAARTHAK MAHAJAN
04496403114

from implementing a framework that enables easy access to computer information. This includes ensuring users have the proper permissions to view, add or manipulate information contained within a system.

Accurate

Garbage in, garbage out (GIGO) is a computer science reference that effectively illustrates how data quality can affect a system's output and can hamper effective decision-making capabilities. A system is only as good as the data you put in it. Verification of the accuracy and completeness of the computer information is crucial to ensuring the computer information meets business needs.

Understandable

Unambiguous and understandable computer information means it is explicit, clear and concise. There is no chance the data can be misinterpreted or misunderstood. On the other hand, ambiguous information can result in multiple interpretations of the same data, which can cause confusion and discord in a system's framework. Even if the information is clarified, the damage to a consumer's perception of the data may be permanent or may take time to reverse.

Valuable

Valuable computer information is tied closely with an organization's business objectives and drivers, so much so that many organizations are demonstrating value by classifying and treating their computer information as strategic assets and employing information assurance techniques to protect that information. The value of computer information can be measured based on an entity's reliance on such information or how much an entity is willing to pay for information.

1.2 INFORMATION SECURITY

Information security, sometimes shortened to InfoSec, is the practice of preventing unauthorized access, use, disclosure, disruption, modification, inspection, recording or destruction of information. It is a general term that can be used regardless of the form the data may take (e.g. electronic, physical).

SAARTHAK MAHAJAN
04496403114

Sometimes referred to as computer security, information technology security (IT security) is information security applied to technology (most often some form of computer system). It is worthwhile to note that a computer does not necessarily mean a home desktop. A computer is any device with a processor and some memory. Such devices can range from non-networked standalone devices as simple as calculators, to networked mobile computing devices such as smartphones and tablet computers. IT security specialists are almost always found in any major enterprise/establishment due to the nature and value of the data within larger businesses. They are responsible for keeping all of the technology within the company secure from malicious cyber-attacks that often attempt to breach into critical private information or gain control of the internal systems.

Information assurance

The act of providing trust of the information, that the Confidentiality, Integrity and Availability (CIA) of the information are not violated, e.g. ensuring that data is not lost when critical issues arise. These issues include, but are not limited to: natural disasters, computer/server malfunction or physical theft. Since most information is stored on computers in our modern era, information assurance is typically dealt with by IT security specialists. A common method of providing information assurance is to have an off-site backup of the data in case one of the mentioned issues arise.

Threats

Information security threats come in many different forms. Some of the most common threats today are software attacks, theft of intellectual property, identity theft, theft of equipment or information, sabotage, and information extortion. Most people have experienced software attacks of some sort. Viruses, worms, phishing attacks, and Trojan horses are a few common examples of software attacks. The theft of intellectual property has also been an extensive issue for many businesses in the IT field. Identity theft is the attempt to act as someone else usually to obtain that person's personal information or to take advantage of their access to vital information. Theft of equipment or information is becoming more prevalent today due to the fact that most devices today are mobile. Cell phones are prone to theft and have also become far more desirable as the amount of data capacity increases. Sabotage usually consists of the destruction of an organization′s website in an attempt to cause loss of confidence on the part of its customers. Information extortion consists of theft of a company′s property or information as an attempt to receive a payment in exchange for returning the information or property back to its owner, as with ransomware. There are many ways to help protect yourself from some of these attacks but one of the most functional precautions is user carefulness.

Governments, military, corporations, financial institutions, hospitals and private businesses amass a great deal of confidential information about their employees, customers, products, research and financial status. Most of this information is now

SAARTHAK MAHAJAN
04496403114

collected, processed and stored on electronic computers and transmitted across networks to other computers.

Should confidential information about a business' customers or finances or new product line fall into the hands of a competitor or a black hat hacker, a business and its customers could suffer widespread, irreparable financial loss, as well as damage to the company's reputation. From a business perspective, information security must be balanced against cost; the Gordon-Loeb Model provides a mathematical economic approach for addressing this concern.

For the individual, information security has a significant effect on privacy, which is viewed very differently in various cultures.

The field of information security has grown and evolved significantly in recent years. It offers many areas for specialization, including securing networks and allied infrastructure, securing applications and databases, security testing, information systems auditing, business continuity planning and digital forensics.

Responses to threats

Possible responses to a security threat or risk are:

- reduce/mitigate – implement safeguards and countermeasures to eliminate vulnerabilities or block threats
- assign/transfer – place the cost of the threat onto another entity or organization such as purchasing insurance or outsourcing
- accept – evaluate if cost of countermeasure outweighs the possible cost of loss due to threat
- ignore/reject – not a valid or prudent due-care response

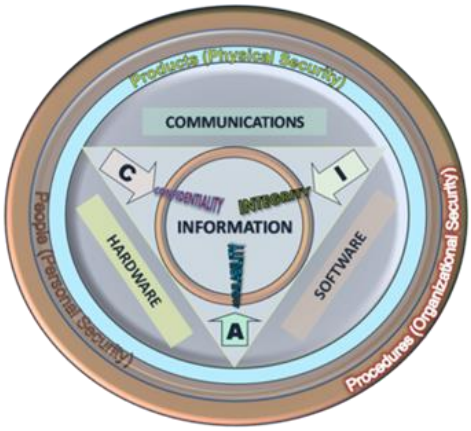


Figure 1.2: Information Security Attributes

SAARTHAK MAHAJAN
04496403114

Key concepts

The CIA triad of confidentiality, integrity, and availability is at the heart of information security. (The members of the classic InfoSec triad — confidentiality, integrity and availability — are interchangeably referred to in the literature as security attributes, properties, security goals, fundamental aspects, information criteria, critical information characteristics and basic building blocks.) There is continuous debate about extending this classic trio. Other principles such as Accountability have sometimes been proposed for addition – it has been pointed out that issues such as non-repudiation do not fit well within the three core concepts.

In 1992 and revised in 2002, the OECD's *Guidelines for the Security of Information Systems and Networks* proposed the nine generally accepted principles: awareness, responsibility, response, ethics, democracy, risk assessment, security design and implementation, security management, and reassessment. Building upon those, in 2004 the NIST's *Engineering Principles for Information Technology Security* proposed 33 principles. From each of these derived guidelines and practices.

In 2002, Donn Parker proposed an alternative model for the classic CIA triad that he called the six atomic elements of information. The elements are confidentiality, possession, integrity, authenticity, availability, and utility. The merits of the Parkerian Hexad are a subject of debate amongst security professionals.

In 2011, The Open Group published the information security management standard O-ISM3. This standard proposed an operational definition of the key concepts of security, with elements called "security objectives", related to access control (9), availability (3), data quality (1), compliance and technical (4). This model is not currently widely adopted.

Confidentiality

In information security, confidentiality "is the property, that information is not made available or disclosed to unauthorized individuals, entities, or processes" (Excerpt ISO27000).

Integrity

In information security, data integrity means maintaining and assuring the accuracy and completeness of data over its entire life-cycle.[21] This means that data cannot be modified in an unauthorized or undetected manner. This is not the same thing as referential integrity in databases, although it can be viewed as a special case of consistency as understood in the classic ACID model of transaction processing. Information security systems typically provide message integrity in addition to data confidentiality.

SAARTHAK MAHAJAN
04496403114

Availability

For any information system to serve its purpose, the information must be available when it is needed. This means that the computing systems used to store and process the information, the security controls used to protect it, and the communication channels used to access it must be functioning correctly. High availability systems aim to remain available at all times, preventing service disruptions due to power outages, hardware failures, and system upgrades. Ensuring availability also involves preventing denial-of-service attacks, such as a flood of incoming messages to the target system essentially forcing it to shut down.

Non-repudiation

In law, non-repudiation implies one's intention to fulfill their obligations to a contract. It also implies that one party of a transaction cannot deny having received a transaction nor can the other party deny having sent a transaction.

It is important to note that while technology such as cryptographic systems can assist in non-repudiation efforts, the concept is at its core a legal concept transcending the realm of technology. It is not, for instance, sufficient to show that the message matches a digital signature signed with the sender's private key, and thus only the sender could have sent the message and nobody else could have altered it in transit (data integrity). The alleged sender could in return demonstrate that the digital signature algorithm is vulnerable or flawed, or allege or prove that his signing key has been compromised. The fault for these violations may or may not lie with the sender himself, and such assertions may or may not relieve the sender of liability, but the assertion would invalidate the claim that the signature necessarily proves authenticity and integrity; and, therefore, the sender may repudiate the message (because authenticity and integrity are pre-requisites for non-repudiation).

1.3 VULNERABILITIES

In computer security, vulnerability is a weakness which allows an attacker to reduce a system's information assurance. Vulnerability is the intersection of three elements: a system susceptibility or flaw, attacker access to the flaw, and attacker capability to exploit the flaw. To exploit a vulnerability, an attacker must have at least one applicable tool or technique that can connect to a system weakness. In this frame, vulnerability is also known as the attack surface.

SAARTHAK MAHAJAN
04496403114

Vulnerability management is the cyclical practice of identifying, classifying, remediating and mitigating vulnerabilities. This practice generally refers to software vulnerabilities in computing systems. Using vulnerability as a method of criminal activity or to create civil unrest falls under US Code Chapter 113B on terrorism
A security risk may be classified as vulnerability. The use of vulnerability with the same meaning of risk can lead to confusion. The risk is tied to the potential of a significant loss. Then there are vulnerabilities without risk: for example when the affected asset has no value.

A vulnerability with one or more known instances of working and fully implemented attacks is classified as an exploitable vulnerability — a vulnerability for which an exploit exists. The window of vulnerability is the time from when the security hole was introduced or manifested in deployed software, to when access was removed, a security fix was available/deployed, or the attackers was disabled—see zero-day attack.

Security bug (security defect) is a narrower concept: there are vulnerabilities that are not related to software: hardware, site, personnel vulnerabilities are examples of vulnerabilities that are not software security bugs.

Constructs in programming languages that are difficult to use properly can be a large source of vulnerabilities.

Vulnerability and risk factor models

A resource (either physical or logical) may have one or more vulnerabilities that can be exploited by a threat agent in a threat action. The result can potentially compromise the confidentiality, integrity or availability of resources (not necessarily the vulnerable one) belonging to an organization and/or other parties involved (customers, suppliers).

An attack can be *active* when it attempts to alter system resources or affect their operation, compromising integrity or availability. A "*passive attack*" attempts to learn or make use of information from the system but does not affect system resources, compromising confidentiality.

OWASP (see figure 1.3) depicts the same phenomenon in slightly different terms: a threat agent through an attack vector exploits a weakness (vulnerability) of the system and the related security controls, causing a technical impact on an IT resource (asset) connected to a business impact. The overall picture represents the risk factors of the risk scenario.

SAARTHAK MAHAJAN
04496403114

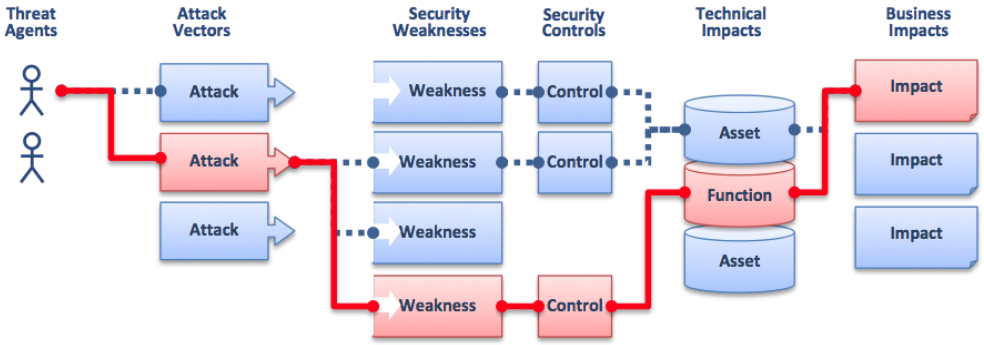


Figure 1.3: OWASP (relationship between threat agent and business impact)

Causes

- Complexity: Large, complex systems increase the probability of flaws and unintended access points.

- Familiarity: Using common, well-known code, software, operating systems, and/or hardware increases the probability an attacker has or can find the knowledge and tools to exploit the flaw.

- Connectivity: More physical connections, privileges, ports, protocols, and services and time each of those are accessible increase vulnerability.

- Password management flaws: The computer user uses weak passwords that could be discovered by brute force. The computer user stores the password on the computer where a program can access it. Users re-use passwords between many programs and websites.

- Fundamental operating system design flaws: The operating system designer chooses to enforce suboptimal policies on user/program management. For example, operating systems with policies such as default permit grant every program and every user full access to the entire computer. This operating system flaw allows viruses and malware to execute commands on behalf of the administrator.

- Internet Website Browsing: Some internet websites may contain harmful Spyware or Adware that can be installed automatically on the computer systems. After visiting those websites, the computer systems become infected and personal information will be collected and passed on to third party individuals.

- Software bugs: The programmer leaves an exploitable bug in a software program. The software bug may allow an attacker to misuse an application.

- Unchecked user input: The program assumes that all user input is safe. Programs that do not check user input can allow unintended direct execution of commands or SQL statements (known as Buffer overflows, SQL injection or other non-validated inputs).

- Not learning from past mistakes: for example most vulnerabilities discovered in IPv4 protocol software were discovered in the new IPv6 implementations.
  The research has shown that the most vulnerable point in most information systems is the human user, operator, designer, or other human: so humans should be considered in their different roles as asset, threat, information resources. Social engineering is an increasing security concern.

Examples of vulnerabilities

Vulnerabilities are related to:

- physical environment of the system
- the personnel
- management
- administration procedures and security measures within the organization
- business operation and service delivery
- hardware
- software
- communication equipment and facilities

It is evident that a pure technical approach cannot even protect physical assets: one should have administrative procedure to let maintenance personnel to enter the facilities and people with adequate knowledge of the procedures, motivated to follow it with proper care.

Four examples of vulnerability exploits:

- an attacker finds and uses an overflow weakness to install malware to export sensitive data;
- an attacker convinces a user to open an email message with attached malware;
- an insider copies a hardened, encrypted program onto a thumb drive and cracks it at home;
- a flood damages one's computer systems installed at ground floor.

SAARTHAK MAHAJAN
04496403114

Software vulnerabilities

Common types of software flaws that lead to vulnerabilities include:

- Memory safety violations, such as:
  - Buffer overflows and over-reads
  - Dangling pointers
- Input validation errors, such as:
  - Format string attacks
  - SQL injection
  - Code injection
  - E-mail injection
  - Directory traversal
  - Cross-site scripting in web applications
  - HTTP header injection
  - HTTP response splitting
- Race conditions, such as:
  - Time-of-check-to-time-of-use bugs
  - Symlink races
- User interface failures, such as:
  - Warning fatigue or user conditioning.
  - Blaming the Victim Prompting a user to make a security decision without giving the user enough information to answer it
  - Race Conditions
- Side-channel attack
  - Timing attack

Many software tools exist that can aid in the discovery (and sometimes removal) of vulnerabilities in a computer system. Though these tools can provide an auditor with a good overview of possible vulnerabilities present, they cannot replace human judgment. Relying solely on scanners will yield false positives and a limited-scope view of the problems present in the system.

1.4 PROGRAM EXECUTION

Execution in computer and software engineering is the process by which a computer or a virtual machine performs the instructions of a computer program. The instructions in the program trigger sequences of simple actions on the executing machine. Those actions produce effects according to the semantics of the instructions in the program.

SAARTHAK MAHAJAN
04496403114

Programs for a computer may execute in a batch process without human interaction, or a user may type commands in an interactive session of an interpreter. In this case the "commands" are simply programs, whose execution is chained together.

The term **run** is used almost synonymously. A related meaning of both "to run" and "to execute" refers to the specific action of a user starting (or *launching* or *invoking*) a program, as in "Please run the application."

Context of execution

The context in which execution takes place is crucial. Very few programs execute on a bare machine. Programs usually contain implicit and explicit assumptions about resources available at the time of execution.

Most programs execute with the support of an operating system and run-time libraries specific to the source language that provide crucial services not supplied directly by the computer itself.

This supportive environment, for instance, usually decouples a program from direct manipulation of the computer peripherals, providing more general, abstract services instead.

Program lifecycle phase

Program lifecycle phases are the stages a computer program undergoes, from initial creation to deployment and execution. The phases are edit time, compile time, distribution time, installation time, link time, load time, and run time.

Lifecycle phases do not necessarily happen in a linear order, and they can be intertwined in various ways. For example, when modifying a program, a software developer may need to repeatedly edit, compile, install, and execute it on his own computer to ensure sufficient quality before it can be distributed to users; copies of the modified program are then downloaded, installed, and executed by users on their computers.

Phases

Edit time (or Design time) is when the source code of the program is being edited. This spans initial creation to any bug fix, refactoring, or addition of new features. Editing is typically performed by a person, but automated design tools and metaprogramming systems may also be used.

SAARTHAK MAHAJAN
04496403114

Compile time is when source code is translated into machine code by a compiler. Part of this involves language checking, such as ensuring proper use of the type system. The result of successful compilation is an executable.

Distribution time is process of transferring a copy of a program to a user. The distribution format is typically an executable, but may also be source code, especially for a program written in an interpreted language. The means of distribution can be physical media such as a USB flash drive or a remote download via the Internet.

Installation time gets the distributed program ready for execution on the user's computer, which often includes storing the executable for future loading by the operating system (OS).

Link time connects all of the necessary machine code components of a program, including externals. It is very common for programs to use functions implemented by external libraries, all of which must be properly linked together. There are two types of linking. Static linking is when the connection is made by the compiler, which is always prior to execution. Dynamic linking, however, is performed by the OS just before, or even during, execution.

Load time is the when the OS takes the program's executable from storage, such as a hard drive, and places it into active memory, in order to begin execution.

Run time is the execution phase, when the central processing unit executes the program's machine code instructions. Programs may run indefinitely. If execution terminates it will either be normal, expected behavior or an abnormality such as a crash.

SAARTHAK MAHAJAN
04496403114

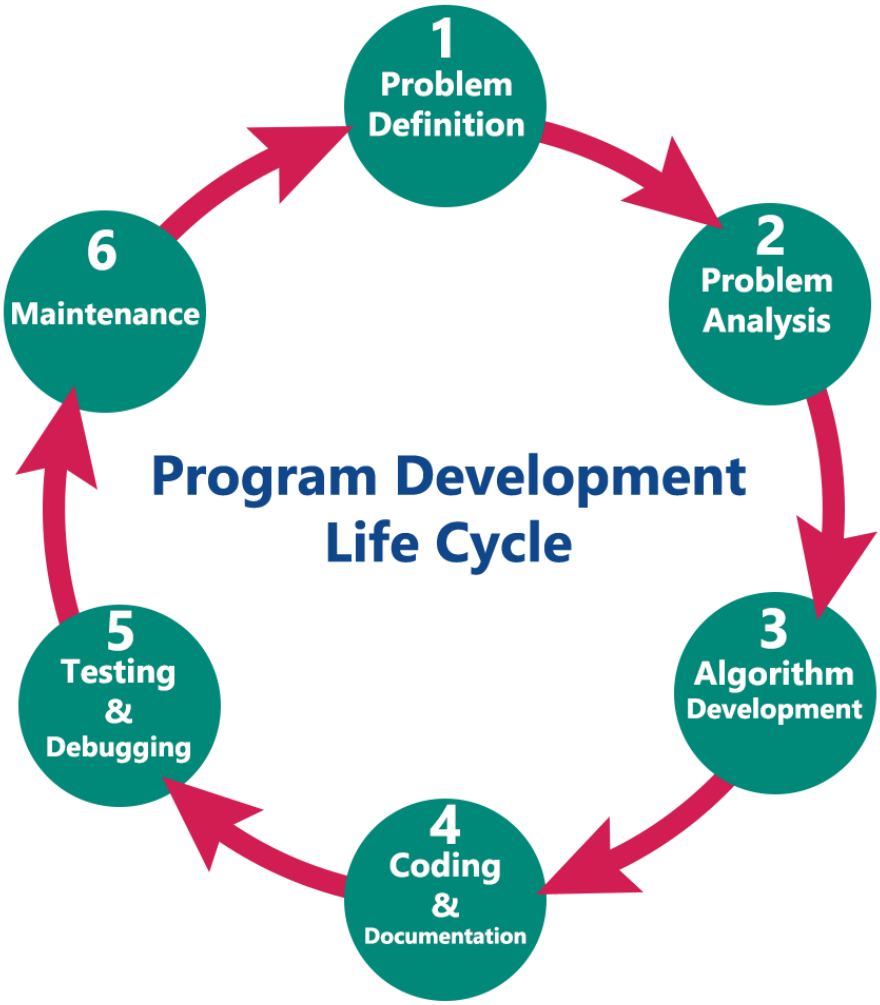


Figure 1.4: Program Lifecycle

1.5 CONTROL FLOW OF A PROGRAM

In computer science, control flow (or flow of control) is the order in which individual statements, instructions or function calls of an imperative program are executed or evaluated. The emphasis on explicit control flow distinguishes an *imperative programming* language from a *declarative programming* language.

Within an imperative programming language, a *control flow statement* is a statement which execution results in a choice being made as to which of two or more paths to follow. For non-strict functional languages, functions and language constructs exist to achieve the same result, but they are usually not termed control flow statements. A set of statements is in turn generally structured as a block, which in addition to grouping, also defines a lexical scope.

SAARTHAK MAHAJAN
04496403114

Interrupts and signals are low-level mechanisms that can alter the flow of control in a way similar to a subroutine, but usually occur as a response to some external stimulus or event (that can occur asynchronously), rather than execution of an *in-line* control flow statement.

At the level of machine language or assembly language, control flow instructions usually work by altering the program counter. For some central processing units (CPUs), the only control flow instructions available are conditional or unconditional branch instructions, also termed jumps.

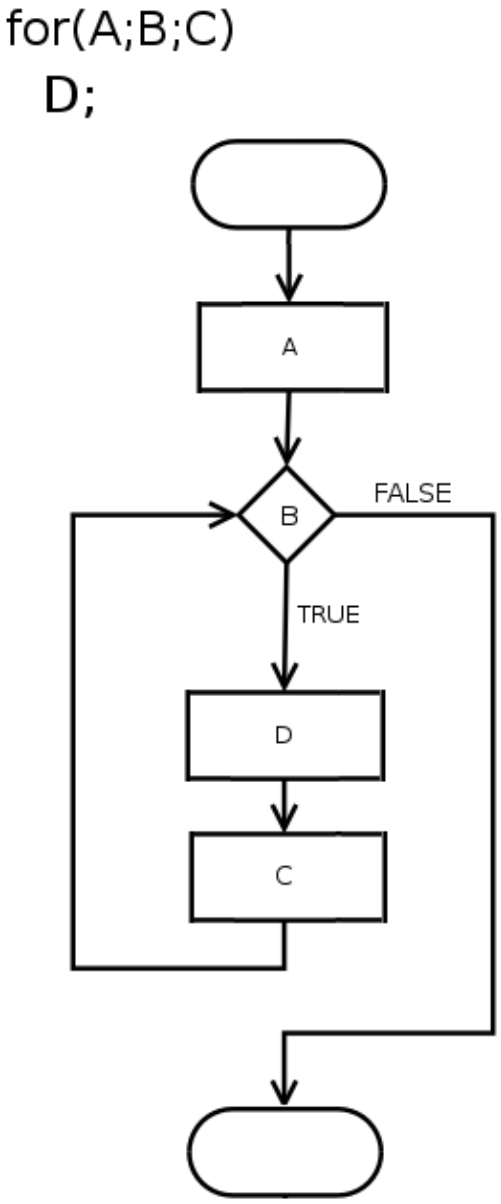


Figure 1.5: FLOW CHART – CONTROL FLOW

SAARTHAK MAHAJAN
04496403114

Categories

The kinds of control flow statements supported by different languages vary, but can be categorized by their effect:

- Continuation at a different statement (unconditional branch or jump)
- Executing a set of statements only if some condition is met (choice - i.e., conditional branch)
- Executing a set of statements zero or more times, until some condition is met (i.e., loop - the same as conditional branch)
- Executing a set of distant statements, after which the flow of control usually returns (subroutines, co-routines, and continuations)
- Stopping the program, preventing any further execution (unconditional halt)

Security

One way to attack a piece of software is to redirect the flow of execution of a program. A variety of control-flow integrity techniques, including stack canaries, buffer overflow protection, shadow stacks, and v-table pointer verification, are used to defend against these attacks.

1.6 CONTROL FLOW INTEGRITY

At a high level, Control-Flow Integrity (CFI) restricts the control-flow of an application to *valid* execution traces. CFI enforces this property by monitoring the program at runtime and comparing its state to a set of precomputed valid states. If an invalid state is detected, an alert is raised, usually terminating the application.

As such, CFI belongs to the class of defenses that leverage *runtime monitors* to detect specific attack vectors (control-flow hijacks for CFI), and then flag exploit attempts at runtime. Other examples of runtime monitors include, e.g., ASan (Address Sanitizer, targeting spatial memory corruption), UBSan (Undefined Behavior Sanitizer, targeting undefined behavior in C/C++), or DangNull (targeting temporal memory corruption). Most of these monitors target development settings, detecting violations when testing the program. CFI, on the other hand, is an active defense mechanism and all modern compilers, e.g., GCC, LLVM, or Microsoft Visual Studio implement a form of CFI with low overhead but different security guarantees.

SAARTHAK MAHAJAN
04496403114

CFI detects control-flow hijacking attacks by limiting the targets of control-flow transfers. In a control-flow hijack attack an attacker redirects the control-flow of the application to locations that would not be reached in a benign execution, e.g., to injected code or to code that is reused in an alternate context.

Since the initial idea for the defense mechanism and the first (closed source) prototype were presented in 2005 a plethora of alternate CFI-style defenses were propose and implemented. While all these alternatives slightly change the underlying enforcement or analysis, they all try to implement the CFI policy. The goal of this blog post is neither to look at differences between individual CFI mechanisms as we did in (see this paper for an exhaustive enumeration of related work in the area of CFI) nor to compare CFI against alternate mechanisms as we did in , but to explain what CFI is, what it can do, and what its limits are.

Any CFI mechanism consists of two abstract components: the (often static) analysis component that recovers the Control-Flow Graph (CFG) of the application (at different levels of precision) and the dynamic enforcement mechanism that restricts control flows according to the generated CFG.

The following sample code shows a simple program with 5 functions. The foo function uses a function pointer and the CFI mechanisms inject both a forward-edge and a backward-edge check. The function pointer either points to bar or Baz. Depending on the forward-edge analysis, different sets of targets are allowed at runtime.

Example -

```
void bar();
void baz();
void buz();
void bez(int, int);

void foo(int usr) {
  void (*func)();

  // func either points to bar or baz
  if (usr == MAGIC)
    func = bar;
  else
    func = baz;

  // forward edge CFI check
  // depending on the precision of CFI:
  // a) all functions {bar, baz, buz, bez, foo} are allowed
```


SAARTHAK MAHAJAN
04496403114

```
  // b) all functions with prototype "void (*)()" are allowed,
  //    i.e., {bar, baz, buz}
  // c) only address taken functions are allowed, i.e., {bar, baz}
  CHECK_CFI_FORWARD(func);
  func();

  // backward edge CFI check
  CHECK_CFI_BACKWARD();
}
```

Control-Flow Transfer Primer

Instructions on architecture can be grouped into control-flow transfer instructions and computational instructions. Computational instructions are executed in sequence (one after the other), while control-flow transfer instructions change control-flow in a specific way, conditionally or unconditionally redirecting control-flow to a specific (code) location.

Control-flow transfer instructions can be further grouped into direct and indirect control-flow transfer instructions. The target of indirect control-flow transfers depends on the runtime value, e.g., of a register or a memory value, compared to direct control-flow transfers where the target is usually encoded as immediate offset in the instruction itself.

Direct control-flow transfers are straight-forward to protect as, on most architectures, executable code is read-only and therefore the target cannot be modified by an attacker (even with arbitrary memory corruption as the write protection bit must first be disabled). These direct control-flow transfers are therefore protected through the read-only permission of individual memory pages and the protection is enforced (at no overhead) by the underlying hardware (the memory management unit). CFI assumes that executable code is read-only, otherwise an attacker could simply overwrite code and remove the runtime monitors.

Indirect control-flow transfers are further divided into **forward-edge** control-flow transfers and **backward-edge** control-flow transfers. Forward-edge control-flow transfers direct code forward to a new location and are used in indirect jump and indirect call instructions, which are mapped at the source code level to, e.g., switch statements, indirect calls, or virtual calls. The backward-edge is used to return to a location that was used in a forward-edge earlier, e.g., when returning from a function call through a return instruction. For simplicity, we leave interrupts, interrupt returns, and exceptions out of the discussion.

SAARTHAK MAHAJAN
04496403114

Generating Control-Flow Graphs

A CFG is a graph that covers all valid executions of the program. Nodes in the graph are locations of control-flow transfers in the program and edges encode reachable targets. The CFG is an abstract concept and the different existing CFI mechanisms use different approaches to generate the underlying CFGs using both static and dynamic analysis, relying on either the binary and source code.

For forward edges, the CFG generation enumerates all possible targets, often leveraging information from the underlying source language. Switch statements in C/C++ are a good example as the different targets are statically known and the compiler can generate a fixed jump table and emit an indirect jump with a bound check to guarantee that the target used at runtime is one of the valid targets in the switch statement.

For indirect function calls through a function pointer, the underlying analysis becomes more complicated as the target may not be known a-priori. Common source-based analyses use a type-based approach and, looking at the function prototype of the function pointer that is used, enumerate all matching functions. Different CFI mechanisms use different forms of type equality, e.g., any valid function, functions with the same arity (number of arguments), or functions with the same signature (arity and equivalence of argument types). At runtime, any function with matching signature is allowed.

Just looking at function prototypes likely yields several collisions where functions are reachable that may never be called in practice. The analysis therefore over-approximates the valid set of targets. In practice, the compiler can check which functions are **address taken**, i.e., there is a source line that generates the address of the function and stores it. The CFI mechanism may reduce the number of allowed targets by intersecting the sets of equal function prototypes and the set of address taken functions.

For virtual calls, i.e., indirect calls in C++ that depend on the type of the object and the class relationship, the analysis can further leverage the type of the object to restrict the valid functions, e.g., all object constructors have the same signature but only the subset constructors of related classes are feasible.

So far, the constructed CFG is stateless, i.e., the *context of the execution* is not considered and each control-flow transfer is independent of all others. On one hand, at runtime only one target is allowed for any possible transfer, namely the target address currently stored at the memory location of the code pointer. CFG construction, on the other hand, over-approximates the number of valid targets with different granularities,

SAARTHAK MAHAJAN
04496403114

depending on the precision of the analysis. Some mechanisms take path constraints into consideration and check (for a limited depth) if the path taken through the application is feasible by using a dynamic analysis approach that validates the current execution. So far, only few mechanisms look at the path context as this incurs dynamic tracking costs at runtime.

Enforcing CFI

CFI can be enforced at different levels. Sometimes the analysis phase (CFG construction) and enforcement phase even overlap, e.g., when considering path constraints. Most mechanisms have two fundamental mechanisms, one for forward-edge transfers and one for backward-edge transfers.

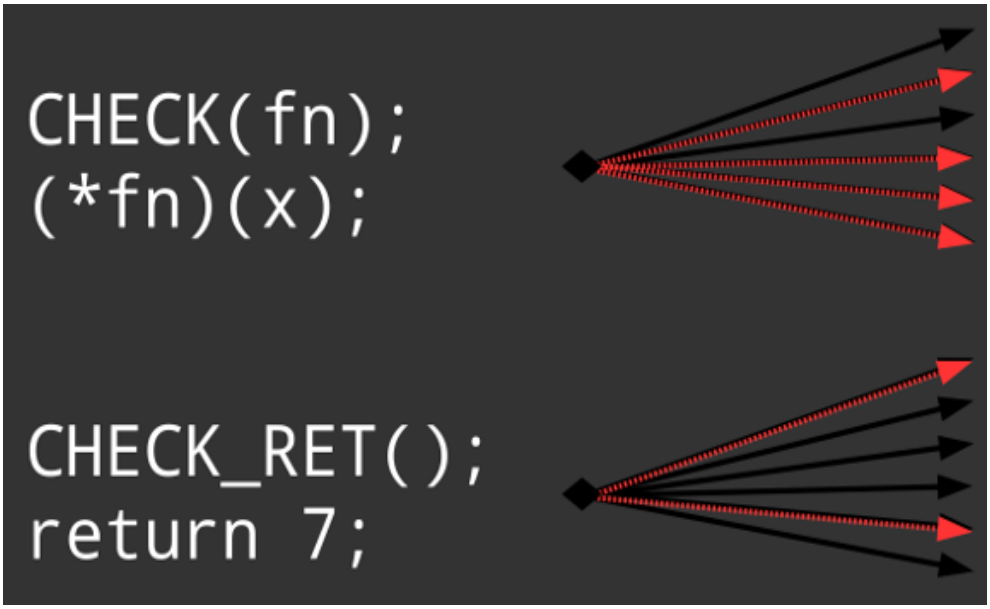


Figure 1.6: CFI validating Targets

The figure shows how CFI restricts the set of possible target locations by executing a runtime monitor that validates the target according to the constructed set of allowed targets. If the observed target is not in that set, the program terminates.

For forward-edge transfers, the code is often instrumented with some form of equivalence check. The check ensures that the target observed at runtime is in the set of valid targets. This can be done through a full set check or a simple type comparison that, e.g., hashes function prototypes and checks if the hash for the current target equals the expected hash at the call site. The hash for the function can be embedded inline in the function, before the function, or in an orthogonal metadata table.

Backward-edge transfers are harder to protect as, when using the same approach, the attacker may redirect the control-flow to any valid call-site when returning from a function. Strong backward-edge protections therefore leverage the context through the previously called functions on the stack.

A mechanism that enforces stack integrity ensures that any backward-edge transfers can only return to the most recent prior caller. This property can be enforced by storing the prior call sites in a shadow stack or guaranteeing memory safety on the stack, i.e., if the return instructions cannot be modified then stack integrity trivially holds.

Backward-edge control-flow enforcement is "easier" than forward-edge, as the function calls and returns form a symbiotic relationship that can be leveraged in the design of the defense, i.e., a function return always returns to the location of the previous call. Such a relationship does not exist for the forward-edge.

Summary

If implemented correctly, CFI is a strong defense mechanism that restricts the freedom of an attacker. Attackers may still corrupt memory and data-only attacks are still in scope.

For the forward-edge, a strong mechanism must consider language-specific semantics to restrict the set of valid targets as much as possible. Additionally, most mechanisms for the forward-edge are stateless and allow an attacker to redirect control-flow to any valid location as identified by the CFG construction. Limiting the size of the target sets constrains the attacker on the forward edge.

For the backward edge, a context-sensitive approach that enforces stack integrity guarantees full protection.

CHAPTER 2

PROGRAM COUNTER & CONTROL

- EIP is a register in x86 architectures (32bit). It holds the “extended instruction pointer" for the stack. In other words, it tells the computer where to go next to execute the next command and controls the flow of a program. In keeping with the naming convention, the *Extended Instruction Pointer* register was referred to as EIP. Most often, the correct value isn't merely EIP, but CS: EIP, because the value of EIP remains an *offset* from some starting location which is defined by the code selector.

- We cannot directly access or change the instruction pointer. However, instructions that control program flow, such as calls, jumps, loops, and interrupts, automatically change the instruction pointer.

- The **program counter** (**PC**), commonly called the **instruction pointer** (**IP**) in Intel x86 and Itanium microprocessors, and sometimes called the **instruction address register** (**IAR**), the **instruction counter**, or just part of the instruction sequencer, is a processor register that indicates where a computer is in its program sequence.

- In most processors, the PC is incremented after fetching an instruction, and holds the memory address of ("points to") the next instruction that would be executed. (In a processor where the incrementation precedes the fetch, the PC points to the current instruction being executed.)

- Processors usually fetch instructions sequentially from memory, but **control transfer** instructions change the sequence by placing a new value in the PC. These include branches (sometimes called jumps), subroutine calls, and returns. A transfer that is conditional on the truth of some assertion lets the computer follow a different sequence under different conditions. A branch provides that the next instruction is fetched from somewhere else in memory. A subroutine call not only branches but saves the preceding contents of the PC somewhere. A return retrieves the saved contents of the PC and places it back in the PC, resuming sequential execution with the instruction following the subroutine call.

SAARTHAK MAHAJAN
04496403114

In a typical central processing unit (CPU), the PC is a digital counter (which is the origin of the term "program counter") that may be one of many registers in the CPU hardware. The instruction cycle begins with a **fetch**, in which the CPU places the value of the PC on the address bus to send it to the memory.

The memory responds by sending the contents of that memory location on the data bus. (This is the stored-program computer model, in which executable instructions are stored alongside ordinary data in memory, and handled identically by it). Following the fetch, the CPU proceeds to **execution**, taking some action based on the memory contents that it obtained.

At some point in this cycle, the PC will be modified so that the next instruction executed is a different one (typically, incremented so that the next instruction is the one starting at the memory address immediately following the last memory location of the current instruction).

Like other processor registers, the PC may be a bank of binary latches, each one representing one bit of the value of the PC. The number of bits (the width of the PC) relates to the processor architecture. For instance, a "32-bit" CPU may use 32 bits to be able to address 2^{32} units of memory. If the PC is a binary counter, it may increment when a pulse is applied to its COUNT UP input, or the CPU may compute some other value and load it into the PC by a pulse to its LOAD input.

To identify the current instruction, the PC may be combined with other registers that identify a segment or page. This approach permits a PC with fewer bits by assuming that most memory units of interest are within the current vicinity.

Consequences in machine architecture

Use of a PC that normally increments assume that what a computer does is execute a usually linear sequence of instructions. Such a PC is central to the von Neumann architecture. Thus programmers write a sequential control flow even for algorithms that do not have to be sequential. The resulting "von Neumann bottleneck" led to research into parallel computing, including non-von Neumann or dataflow models that did not use a PC; for example, rather than specifying sequential steps, the high-level programmer might specify desired function and the low-level programmer might specify this using combinatory logic.

SAARTHAK MAHAJAN
04496403114

This research also led to ways to making conventional, PC-based, CPUs run faster, including:

- Pipelining, in which different hardware in the CPU executes different phases of multiple instructions simultaneously.
- The very long instruction word (VLIW) architecture, where a single instruction can achieve multiple effects.
- Techniques to predict out-of-order execution and prepare subsequent instructions for execution outside the regular sequence.

Consequences in high-level programming

Modern high-level programming languages still follow the sequential-execution model and, indeed, a common way of identifying programming errors is with a "procedure execution" in which the programmer's finger identifies the point of execution as a PC would. The high-level language is essentially the machine language of a virtual machine,[9] too complex to be built as hardware but instead emulated or interpreted by software.

However, new programming models transcend sequential-execution programming:

- When writing a multi-threaded program, the programmer may write each thread as a sequence of instructions without specifying the timing of any instruction relative to instructions in other threads.
- In event-driven programming, the programmer may write sequences of instructions to respond to events without specifying an overall sequence for the program.
- In dataflow programming, the programmer may write each section of a computing pipeline without specifying the timing relative to other sections.

In computing, an **instruction register (IR)** is the part of a CPU's control unit that holds the instruction currently being executed or decoded.[1] In simple processors each instruction to be executed is loaded into the instruction register which holds it while it is decoded, prepared and ultimately executed, which can take several steps.

Decoding the op-code in the instruction register includes determining the instruction, determining where its operands are in memory, retrieving the operands from memory, allocating processor resources to execute the command (in superscalar processors), etc.

SAARTHAK MAHAJAN
04496403114

The output of IR is available to control circuits which generate the timing signals that control the various processing elements involved in executing the instruction. In the instruction cycle, the instruction is loaded into the Instruction register after the processor fetches it from the memory location pointed by the program counter.

Figure 2.1: IBM 701 Panel(Instruction Pointer at lower left)

A branch is an instruction in a computer program that can cause a computer to begin executing a different instruction sequence and thus deviate from its default behavior of executing instructions in order. Branch (or *branching*, *branched*) may also refer to the act of switching execution to a different instruction sequence as a result of executing a branch instruction. A branch instruction can be either an *unconditional branch*, which always results in branching, or a *conditional branch*, which may or may not cause branching, depending on some condition. Branch instructions are used to implement control flow in program loops and conditionals (i.e., executing a particular sequence of instructions only if certain conditions are satisfied).

SAARTHAK MAHAJAN
04496403114

Implementation

Mechanically, a branch instruction can change the program counter of a CPU. The program counter is the memory address of the next instruction. Therefore, a branch can cause the CPU to begin fetching its instructions from a different sequence of memory cells.

When a branch is *taken*, the CPU's program counter is set to the argument of the jump instruction. So, the next instruction becomes the instruction at that address in memory. Therefore, the flow of control changes.

When a branch is *not taken*, the CPU's program counter is unchanged. Therefore, the next instruction executed is the instruction after the branch instruction. Therefore, the flow of control is unchanged.

The term *branch* can be used when referring to programs in high level languages as well as program written in machine code or assembly language. In high-level programming languages, branches usually take the form of conditional statements of various forms that encapsulate the instruction sequence that will be executed if the conditions are satisfied. Unconditional branch instructions such as GOTO are used to unconditionally "jump" to (begin execution of) a different instruction sequence.

Machine level branch instructions are sometimes called *jump* instructions. Machine level jump instructions typically have *unconditional* and *conditional* forms where the latter may be *taken* or *not taken* depending on some condition.

In CPUs with flag registers, an earlier instruction sets a condition in the flag register. The earlier instruction may be arithmetic, or a logic instruction. It is often close to the branch, though not necessarily the instruction *immediately* before the branch. The stored condition is then used in a branch such as *jump if overflow-flag set*. This temporary information is often stored in a flag register but may also be located elsewhere. A flag register design is simple in slower, simple computers.

There are also machines (or particular instructions) where the condition may be checked by the jump instruction itself, such as *branch <label> if register X negative*. In simple computer designs, comparison branches execute more arithmetic and can use more power than flag register branches. In fast computer designs comparison branches can run faster than flag register branches, because comparison branches can access the registers with more parallelism, using the same CPU mechanisms as a calculation.

SAARTHAK MAHAJAN
04496403114

CHAPTER 3

MEMORY PROTECTION

3.1 INTRODUCTION

Memory protection is a way to control memory access rights on a computer, and is a part of most modern instruction set architectures and operating systems. The main purpose of memory protection is to prevent a process from accessing memory that has not been allocated to it. This prevents a bug or malware within a process from affecting other processes, or the operating system itself. An attempt to access unowned memory results in a hardware fault, called a segmentation fault or storage violation exception, generally causing abnormal termination of the offending process. Memory protection for computer security includes additional techniques such as address space layout randomization and executable space protection.

Methods

Segmentation

Segmentation refers to dividing a computer's memory into segments. A reference to a memory location includes a value that identifies a segment and an offset within that segment.

The x86 architecture has multiple segmentation features, which are helpful for using protected memory on this architecture. On the x86 architecture, the Global Descriptor Table and Local Descriptor Tables can be used to reference segments in the computer's memory. Pointers to memory segments on x86 processors can also be stored in the processor's segment registers. Initially x86 processors had 4 segment registers, CS (code segment), SS (stack segment), DS (data segment) and ES (extra segment); later another two segment registers were added – FS and GS.

Paged virtual memory

In paging the memory address space is divided into equal-sized blocks called pages. Using virtual memory hardware, each page can reside in any location of the computer's physical memory, or be flagged as being protected. Virtual memory makes it possible to have a linear virtual memory address space and to use it to access blocks fragmented over physical memory address space.

Most computer architectures which support paging also use pages as the basis for memory protection.

SAARTHAK MAHAJAN
04496403114

A *page table* maps virtual memory to physical memory. The page table is usually invisible to the process. Page tables make it easier to allocate additional memory, as each new page can be allocated from anywhere in physical memory.

It is impossible for an application to access a page that has not been explicitly allocated to it, because every memory address either points to a page allocated to that application, or generates an interrupt called a *page fault*. Unallocated pages, and pages allocated to any other application, do not have any addresses from the application point of view.

A page fault may not necessarily indicate an error. Page faults are not only used for memory protection. The operating system may manage the page table in such a way that a reference to a page that has been previously swapped out to disk causes a page fault. The operating system intercepts the page fault, loads the required memory page, and the application continues as if no fault had occurred. This scheme, known as virtual memory, allows in-memory data not currently in use to be moved to disk storage and back in a way which is transparent to applications, to increase overall memory capacity.

On some systems, the page fault mechanism is also used for executable space protection such as W^X.

Protection keys

A memory protection key (MPK) mechanism divides physical memory up into blocks of a particular size (e.g., 4 kB), each of which has an associated numerical value called a protection key. Each process also has a protection key value associated with it. On a memory access the hardware checks that the current process's protection key matches the value associated with the memory block being accessed; if not, an exception occurs. This mechanism was introduced in the System/360 architecture. It is available on today's System z mainframes and heavily used by System z operating systems and their subsystems.

The System/360 protection keys described above are associated with physical addresses. This is different from the protection key mechanism used by architectures such as the Hewlett-Packard/Intel IA-64 and Hewlett-Packard PA-RISC, which are associated with virtual addresses, and which allow multiple keys per process.

In the Itanium and PA architectures, translations (TLB entries) have *keys* (Itanium) or *access ids* (PA) associated with them. A running process has several protection key registers (16 for Itanium 4 for HP/PA). A translation selected by the virtual address has its key compared to each of the protection key registers. If any of them match (plus other possible checks), the access is permitted. If none match, a fault or exception is generated. The software fault handler can, if desired, check the missing key against a larger list of keys maintained by software; thus, the protection key

SAARTHAK MAHAJAN
04496403114

registers inside the processor may be treated as a software-managed cache of a larger list of keys associated with a process.

PA has 15–18 bits of key; Itanium mandates at least 18. Keys are usually associated with *protection domains*, such as libraries, modules, etc.

In the x86, the protection keys architecture allows tagging virtual addresses for user pages with any of 16 protection keys. All the pages tagged with the same protection key constitute a protection domain, A new register contains the permissions associated with each of the protection domain. Load and store operations are checked against both the page table permissions and the protection key permissions associated with the protection domain of the virtual address, and only allowed if both permissions allow the access. The protection key permissions can be set from user space, allowing applications to directly restrict access to the application data without OS intervention. Since the protection keys are associated with a virtual address, the protection domains are per address space, so processes running in different address spaces can each use all 16 domains.

Simulated segmentation

Simulation is use of a monitoring program to interpret the machine code instructions of some computer architectures. Such an instruction set simulator can provide memory protection by using a segmentation-like scheme and validating the target address and length of each instruction in real time before actually executing them. The simulator must calculate the target address and length and compare this against a list of valid address ranges that it holds concerning the thread's environment, such as any dynamic memory blocks acquired since the thread's inception, plus any valid shared static memory slots. The meaning of "valid" may change throughout the thread's life depending upon context. It may sometimes be allowed to alter a static block of storage, and sometimes not, depending upon the current mode of execution, which may or may not depend on a storage key or supervisor state.

It is generally not advisable to use this method of memory protection where adequate facilities exist on a CPU, as this takes valuable processing power from the computer. However, it is generally used for debugging and testing purposes to provide an extra fine level of granularity to otherwise generic storage violations and can indicate precisely which instruction is attempting to overwrite the particular section of storage which may have the same storage key as unprotected storage.

Capability-based addressing

Capability-based addressing is a method of memory protection that is unused in modern commercial computers. In this method, pointers are replaced by protected objects (called *capabilities*) that can only be created via using privileged instructions which may only be executed by the kernel, or some other process authorized to do so. This effectively lets the kernel control which processes may access which objects in

SAARTHAK MAHAJAN
04496403114

memory, with no need to use separate address spaces or context switches. Only a few commercial products used capability based security: Plessey System 250, IBM System/38, Intel iAPX 432 architecture and KeyKOS. Capability approaches are widely used in research systems such as EROS and Combex DARPA Browser. They are used conceptually as the basis for some virtual machines, most notably Smalltalk and Java. Currently, the DARPA-funded CHERI project at University of Cambridge is working to create a modern capability machine that also supports legacy software.

Dynamic tainting

Dynamic tainting is a technique for protecting programs from illegal memory accesses. When memory is allocated, at runtime, this technique taints both the memory and the corresponding pointer using the same taint mark. Taint marks are then suitably propagated while the program executes and are checked every time a memory address *m* is accessed through a pointer *p*; if the taint marks associated with *m* and *p* differ, the execution is stopped and the illegal access is reported.[6]

SPARC M7 processors (and higher) implement dynamic tainting in hardware. This is called Silicon Secured Memory (SSM) or Application Data Integrity (ADI).

Different operating systems use different forms of memory protection or separation. Although memory protection was common on most mainframes and many minicomputer systems from the 1960s, true memory separation was not used in home computer operating systems until OS/2 was released in 1987. On prior systems, such lack of protection was even used as a form of interprocess communication, by sending a pointer between processes. It is possible for processes to access System Memory in the Windows 9x family of Operating Systems.

Some operating systems that do implement memory protection include:

- Unix-like systems (since the late '70s), including Solaris, Linux, BSD, macOS, iOS and GNU Hurd
- Plan9 and Inferno, created at Bell Labs as Unix successors (1992, 1995).
- OS/2 (1987)

3.2 ADDRESS SPACE LAYOUT RANDOMIZATION

Address space layout randomization (ASLR) is a computer security technique involved in preventing exploitation of memory corruption vulnerabilities. In order to prevent an attacker from reliably jumping to, for example, a particular exploited function in memory, ASLR randomly arranges the address space positions of key data areas of a process, including the base of the executable and the positions **of the stack, heap and libraries.**

SAARTHAK MAHAJAN
04496403114

History

The Linux PaX project first coined the term "ASLR", and published the first design and implementation of ASLR in July 2001 as a patch for the Linux kernel. It is seen as a complete implementation, providing also a patch for kernel stack randomization since October 2002.

The first mainstream operating system to support ASLR by default was the OpenBSD version 3.4 in 2003.

Address space randomization hinders some types of security attacks by making it more difficult for an attacker to predict target addresses. For example, attackers trying to execute return-to-libc attacks must locate the code to be executed, while other attackers trying to execute shellcode injected on the stack have to find the stack first. In both cases, the system obscures related memory-addresses from the attackers. These values have to be guessed, and a mistaken guess is not usually recoverable due to the application crashing.

Effectiveness

Address space layout randomization is based upon the low chance of an attacker guessing the locations of randomly placed areas. Security is increased by increasing the search space. Thus, address space randomization is more effective when more entropy is present in the random offsets. Entropy is increased by either raising the amount of virtual memory area space over which the randomization occurs or reducing the period over which the randomization occurs. The period is typically implemented as small as possible, so most systems must increase VMA space randomization.

To defeat the randomization, attackers must successfully guess the positions of all areas they wish to attack. For data areas such as stack and heap, where custom code or useful data can be loaded, more than one state can be attacked by using NOP slides for code or repeated copies of data. This allows an attack to succeed if the area is randomized to one of a handful of values. In contrast, code areas such as library base and main executable need to be discovered exactly. Often these areas are mixed, for example stack frames are injected onto the stack and a library is returned into.

SAARTHAK MAHAJAN
04496403114

Implementations

Several mainstream, general-purpose operating systems implement ASLR.

Android

Android 4.0 Ice Cream Sandwich provides address space layout randomization (ASLR) to help protect system and third party applications from exploits due to memory-management issues. Position-independent executable support was added in Android 4.1. Android 5.0 dropped non-PIE support and requires all dynamically linked binaries to be position independent. Library load ordering randomization was accepted into the Android open-source project on 26 October 2015, and was included in the Android 7.0 release.

DragonFly BSD

DragonFly BSD has an implementation of ASLR based upon OpenBSD's model, added in 2010. It is off by default, and can be enabled by setting the sysctl vm.randomize_mmap to 1.

iOS (iPhone, iPod touch, iPad)

Apple introduced ASLR in iOS 4.3 (released March 2011).

KASLR randomized Kernel base = 0x01000000 + ((1+0xRR) * 0x00200000)

where 0xRR = a random byte from SHA1(random data) generated by iBoot (the 2nd-stage iOS Boot Loader)

Linux

Linux kernel enabled a weak form of ASLR by default since the kernel version 2.6.12, released in June 2005. The PaX and Exec Shield patchsets to the Linux kernel provide more complete implementations. The Exec Shield patch for Linux supplies 19 bits of stack entropy on a period of 16 bytes, and 8 bits of mmap base randomization on a period of 1 page of 4096 bytes. This places the stack base in an area 8 MB wide containing 524 288 possible positions, and the mmap base in an area 1 MB wide containing 256 possible positions. Various Linux distributions—including

SAARTHAK MAHAJAN
04496403114

Adamantix, Alpine Linux, Hardened Gentoo, and Hardened Linux From Scratch—come with PaX's implementation of ASLR by default.

Position-independent executable (PIE) implements a random base address for the main executable binary and has been in place since 2003. It provides the same address randomness to the main executable as being used for the shared libraries. The PIE feature is in use only for the network facing daemon – the PIE feature cannot be used together with the prelink feature for the same executable. The prelink tool implements randomization at prelink time rather than runtime, because by design prelink aims to handle relocating libraries before the dynamic linker has to, which allows the relocation to occur once for many runs of the program. As a result, real address space randomization would defeat the purpose of prelinking.

Kernel address space layout randomization (KASLR), bringing support for address space randomization to running Linux kernel images by randomizing where the kernel code is placed at boot time, was merged into the Linux kernel mainline in kernel version 3.14, released on 30 March 2014. When compiled in, it can be disabled at boot time by specifying nokaslr as one of the kernel's boot parameters. KASLR was sharply criticized by Brad Spengler, primary developer of grsecurity, to be providing very limited additional levels of security.

Microsoft Window

Microsoft's Windows Vista (released January 2007) and later have ASLR enabled for only those executables and dynamic link libraries specifically linked to be ASLR-enabled. For compatibility, it is not enabled by default for other applications. Typically, only older software is incompatible and ASLR can be fully enabled by editing a registry entry "HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management\MoveImages", or by installing Microsoft's Enhanced Mitigation Experience Toolkit.

The locations of the heap, stack, Process Environment Block, and Thread Environment Block are also randomized. A security whitepaper from Symantec noted that ASLR in 32-bit Windows Vista may not be as robust as expected, and Microsoft has acknowledged a weakness in its implementation.

Host-based intrusion prevention systems such as *WehnTrust* and *Ozone* also offer ASLR for Windows XP and Windows Server 2003 operating systems. WehnTrust is open-source. Complete details of Ozone's implementation is not available.

It was noted in February 2012 that ASLR on 32-bit Windows systems prior to Windows 8 can have its effectiveness reduced in low memory situations. Similar effect also had been achieved on Linux in the same research. The test code caused the Mac OS X 10.7.3 system to kernel panic, so it was left unclear about its ASLR behavior in this scenario.

NetBSD

Support for ASLR appeared in NetBSD 5.0 (released April 2009), and was enabled by default in NetBSD-current in April 2016.

OpenBSD

In 2003, OpenBSD became the first mainstream operating system to support a strong form of ASLR and to activate it by default. OpenBSD completed its ASLR support in 2008 when it added support for PIE binaries. OpenBSD 4.4's malloc(3) was designed to improve security by taking advantage of ASLR and gap page features implemented as part of OpenBSD's mmap system call, and to detect use-after-free bugs. Released in 2013, OpenBSD 5.3 was the first mainstream operating system to enable Position-independent executables by default on multiple hardware platforms, and OpenBSD 5.7 activated position-independent static binaries (Static-PIE) by default.

OS X

In Mac OS X Leopard 10.5 (released October 2007), Apple introduced randomization for system libraries.

In Mac OS X Lion 10.7 (released July 2011), Apple expanded their implementation to cover all applications, stating "address space layout randomization (ASLR) has been improved for all applications. It is now available for 32-bit apps (as are heap memory protections), making 64-bit and 32-bit applications more resistant to attack."

As of OS X Mountain Lion 10.8 (released July 2012) and later, the entire system including the kernel as well as kexts and zones are randomly relocated during system boot.

3.3 DATA EXECUTION PREVENTION

In computer security, **executable-space protection** marks memory regions as non-executable, such that an attempt to execute machine code in these regions will cause an exception. It makes use of hardware features such as the NX bit (no-execute bit), or in some cases software emulation of those features. However technologies that somehow emulate or supply an NX bit will usually impose a measurable overhead; while using a hardware-supplied NX bit imposes no measurable overhead.

The Burroughs 5000 offered hardware support for executable-space protection on its introduction in 1961; that capability remained in its successors until at least 2006. In its implementation of tagged architecture, each word of memory had an associated, hidden tag bit designating it code or data. Thus user programs cannot write or even read a program word, and data words cannot be executed.

If an operating system can mark some or all writable regions of memory as non-executable, it may be able to prevent the stack and heap memory areas from being executable. This helps to prevent certain buffer-overflow exploits from succeeding, particularly those that inject and execute code, such as the Sasser and Blaster worms. These attacks rely on some part of memory, usually the stack, being both writeable and executable; if it is not, the attack fails.

- It marks memory regions as non-executable, such that an attempt to execute machine code in these regions will cause an exception. It makes use of hardware features such as the NX bit (no-execute bit).
- DEP/NX alone does not provide perfect protection against all memory-based vulnerabilities, but as with ASLR technologies, it can provide advanced protection on memory based vulnerabilities.
- It stops an attacker from being able to directly execute code from the stack, heap, and other non-code memory regions.

OS implementations

Many operating systems implement or have an available executable space protection policy. Here is a list of such systems in alphabetical order, each with technologies ordered from newest to oldest.

For some technologies, there is a summary which gives the major features each technology supports. The summary is structured as below.

- Hardware Supported Processors: (Comma separated list of CPU architectures)
- Emulation: (No) or (Architecture Independent) or (Comma separated list of CPU architectures)
- Other Supported: (None) or (Comma separated list of CPU architectures)

SAARTHAK MAHAJAN
04496403114

- Standard Distribution: (No) or (Yes) or (Comma separated list of distributions or versions which support the technology)
- Release Date: (Date of first release)

A technology supplying Architecture Independent emulation will be functional on all processors which aren't hardware supported. The "Other Supported" line is for processors which allow some grey-area method, where an explicit NX bit doesn't exist yet hardware allows one to be emulated in some way.

Android

As of Android 2.3 and later, architectures which support it have non-executable pages by default, including non-executable stack and heap.

FreeBSD

Initial support for the NX bit, on x86-64 and IA-32 processors that support it, first appeared in FreeBSD -CURRENT on June 8, 2004. It has been in FreeBSD releases since the 5.3 release.

Linux

The Linux kernel supports the NX bit on x86-64 and IA-32 processors that support it, such as modern 64-bit processors made by AMD, Intel, Transmeta and VIA. The support for this feature in the 64-bit mode on x86-64 CPUs was added in 2004 by Andi Kleen, and later the same year, Ingo Molnar added support for it in 32-bit mode on 64-bit CPUs. These features have been part of the Linux kernel mainline since the release of kernel version 2.6.8 in August 2004.

The availability of the NX bit on 32-bit x86 kernels, which may run on both 32-bit x86 CPUs and 64-bit IA-32-compatible CPUs, is significant because a 32-bit x86 kernel would not normally expect the NX bit that an AMD64 or IA-64 supplies; the NX enabler patch assures that these kernels will attempt to use the NX bit if present.

Some desktop Linux distributions, such as Fedora, Ubuntu and openSUSE, do not enable the HIGHMEM64 option by default in their default kernels, which is required to gain access to the NX bit in 32-bit mode, because the PAE mode that is required to use the NX bit causes boot failures on pre-Pentium Pro (including Pentium MMX) and Celeron M and Pentium M processors without NX support. Other processors that do not support PAE are AMD K6 and earlier, Transmeta Crusoe, VIA C3 and earlier, and Geode GX and LX. VMware Workstation versions older than 4.0, Parallels Workstation versions older than 4.0, and Microsoft Virtual PC and Virtual Server do not support PAE on the guest. Fedora Core 6 and Ubuntu 9.10 and later provide a kernel-PAE package which supports PAE and NX.

SAARTHAK MAHAJAN
04496403114

NX memory protection has always been available in Ubuntu for any systems that had the hardware to support it and ran the 64-bit kernel or the 32-bit server kernel. The 32-bit PAE desktop kernel (linux-image-generic-pae) in Ubuntu 9.10 and later, also provides the PAE mode needed for hardware with the NX CPU feature. For systems that lack NX hardware, the 32-bit kernels now provide an approximation of the NX CPU feature via software emulation that can help block many exploits an attacker might run from stack or heap memory.

Non-execute functionality has also been present for other non-x86 processors supporting this functionality for many releases.

PaX

The PaX NX technology can emulate NX functionality, or use a hardware NX bit. PaX works on x86 CPUs that do not have the NX bit, such as 32-bit x86. The Linux kernel still does not ship with PaX (as of May, 2007); the patch must be merged manually.

PaX provides two methods of NX bit emulation, called SEGMEXEC and PAGEEXEC. The SEGMEXEC method imposes a measurable but low overhead, typically less than 1%, which is a constant scalar incurred due to the virtual memory mirroring used for the separation between execution and data accesses.[5] SEGMEXEC also has the effect of halving the task's virtual address space, allowing the task to access less memory than it normally could. This is not a problem until the task requires access to more than half the normal address space, which is rare. SEGMEXEC does not cause programs to use more system memory (i.e. RAM), it only restricts how much they can access. On 32-bit CPUs, this becomes 1.5 GB rather than 3 GB.

PaX supplies mprotect() restrictions to prevent programs from marking memory in ways that produce memory useful for a potential exploit. This policy causes certain applications to cease to function, but it can be disabled for affected programs.

PaX allows individual control over the following functions of the technology for each binary executable:

- PAGEEXEC
- SEGMEXEC
- mprotect() restrictions
- Trampoline emulation
- Randomized executable base
- Randomized mmap() base

SAARTHAK MAHAJAN
04496403114

- Hardware Supported Processors: Alpha, AMD64, IA-64, MIPS (32 and 64 bit), PA-RISC, PowerPC, SPARC
- Emulation: IA-32 (x86)
- Other Supported: PowerPC (32 and 64 bit), SPARC (32 and 64 bit)
- Standard Distribution: Adamantix, Hardened Gentoo, Hardened Linux, Alpine Linux
- Release Date: October 1, 2000

NetBSD

As of NetBSD 2.0 and later (December 9, 2004), architectures which support it have non-executable stack and heap.

Architectures that have per-page granularity consist of: alpha, amd64, hppa, i386 (with PAE), powerpc (ibm4xx), sh5, sparc (sun4m, sun4d), sparc64.

Architectures that can only support these with region granularity are: i386 (without PAE), other powerpc (such as macppc).

Other architectures do not benefit from non-executable stack or heap; NetBSD does not by default use any software emulation to offer these features on those architectures.

OpenBSD

A technology in the OpenBSD operating system, known as W^X, marks writable pages by default as non-executable on processors that support that. On 32-bit x86 processors, the code segment is set to include only part of the address space, to provide some level of executable space protection.

OpenBSD 3.3 shipped May 1, 2003, and was the first to include W^X.

- Hardware Supported Processors: Alpha, AMD64, HPPA, SPARC
- Emulation: IA-32 (x86)
- Other Supported: None
- Standard Distribution: Yes
- Release Date: May 1, 2003

SAARTHAK MAHAJAN
04496403114

macOS

macOS for Intel supports the NX bit on all CPUs supported by Apple (from Mac OS X 10.4.4 – the first Intel release – onwards). Mac OS X 10.4 only supported NX stack protection. In Mac OS X 10.5, all 64-bit executables have NX stack and heap; W^X protection. This includes x86-64 (Core 2 or later) and 64-bit PowerPC on the G5 Macs.

Solaris

Solaris has supported globally disabling stack execution on SPARC processors since Solaris 2.6 (1997); in Solaris 9 (2002), support for disabling stack execution on a per-executable basis was added.

Windows

Starting with Windows XP Service Pack 2 (2004) and Windows Server 2003 Service Pack 1 (2005), the NX features were implemented for the first time on the x86 architecture. Executable space protection on Windows is called "Data Execution Prevention" (DEP).

Under Windows XP or Server 2003 NX protection was used on critical Windows services exclusively by default. If the x86 processor supported this feature in hardware, then the NX features were turned on automatically in Windows XP/Server 2003 by default. If the feature was not supported by the x86 processor, then no protection was given.

Early implementations of DEP provided no address space layout randomization (ASLR), which allowed potential return-to-libc attacks that could have been feasibly used to disable DEP during an attack. The PaX documentation elaborates on why ASLR is necessary; a proof-of-concept was produced detailing a method by which DEP could be circumvented in the absence of ASLR. It may be possible to develop a successful attack if the address of prepared data such as corrupted images or MP3s can be known by the attacker.

Microsoft added ASLR functionality in Windows Vista and Windows Server 2008. On this platform, DEP is implemented through the automatic use of PAE kernel in 32-bit Windows and the native support on 64-bit kernels. Windows Vista DEP works by marking certain parts of memory as being intended to hold only data, which the NX or XD bit enabled processor then understands as non-executable. In Windows, from version Vista, whether DEP is enabled or disabled for a particular process can be viewed on the *Processes* tab in the Windows Task Manager.

Windows implements software DEP (without the use of the NX bit) through Microsoft's "Safe Structured Exception Handling" (SafeSEH). For properly compiled applications, SafeSEH checks that, when an exception is raised during program

SAARTHAK MAHAJAN
04496403114

execution, the exception's handler is one defined by the application as it was originally compiled. The effect of this protection is that an attacker is not able to add his own exception handler which he has stored in a data page through unchecked program input.

When NX is supported, it is enabled by default. Windows allows programs to control which pages disallow execution through its API as well as through the section headers in a PE file. In the API, runtime access to the NX bit is exposed through the Win32 API calls **VirtualAlloc[Ex]** and **VirtualProtect[Ex]**. Each page may be individually flagged as executable or non-executable. Despite the lack of previous x86 hardware support, both executable and non-executable page settings have been provided since the beginning. On pre-NX CPUs, the presence of the 'executable' attribute has no effect. It was documented as if it did function, and, as a result, most programmers used it properly. In the PE file format, each section can specify its executability. The execution flag has existed since the beginning of the format and standard linkers have always used this flag correctly, even long before the NX bit. Because of this, Windows is able to enforce the NX bit on old programs. Assuming the programmer complied with "best practices", applications should work correctly now that NX is actually enforced. Only in a few cases have there been problems; Microsoft's own .NET Runtime had problems with the NX bit and was updated.

- Hardware Supported Processors: x86-64 (AMD64 and Intel 64), IA-64, Efficeon, Pentium M (later revisions), AMD Sempron (later revisions)
- Emulation: Yes
- Other Supported: None
- Standard Distribution: Post Windows XP
- Release Date: August 6, 2004

LIMITATIONS

- Many of today's applications do not support ASLR because those applications need to be specially linked to take advantage of this feature. Applications that support ASLR use slightly more RAM than those that don't.
- Older operating systems (e.g., Windows XP) do not support DEP.
- ASLR & DEP are most effective when used together.
- Together they are not a silver bullet. Memory module mapping can bypass ALSR, and Return-Oriented Programming (ROP) chaining can bypass DEP.

SAARTHAK MAHAJAN
04496403114

CHAPTER 4

RETURN ORIENTED PROGRAMMING

Return-oriented programming (ROP) is a computer security exploit technique that allows an attacker to execute code in the presence of security defences such as non-executable memory (W or X technique) and code signing.

In this technique, an attacker gains control of the call stack to hijack program control flow and then executes carefully chosen machine instruction sequences that are already presented in machine's memory, called "gadgets". Each gadget typically ends in a return instruction and is located in a subroutine within the existing program and/or shared library code.

Chained together, these gadgets allow an attacker to perform arbitrary operations on a machine employing defences that thwart simpler attacks.

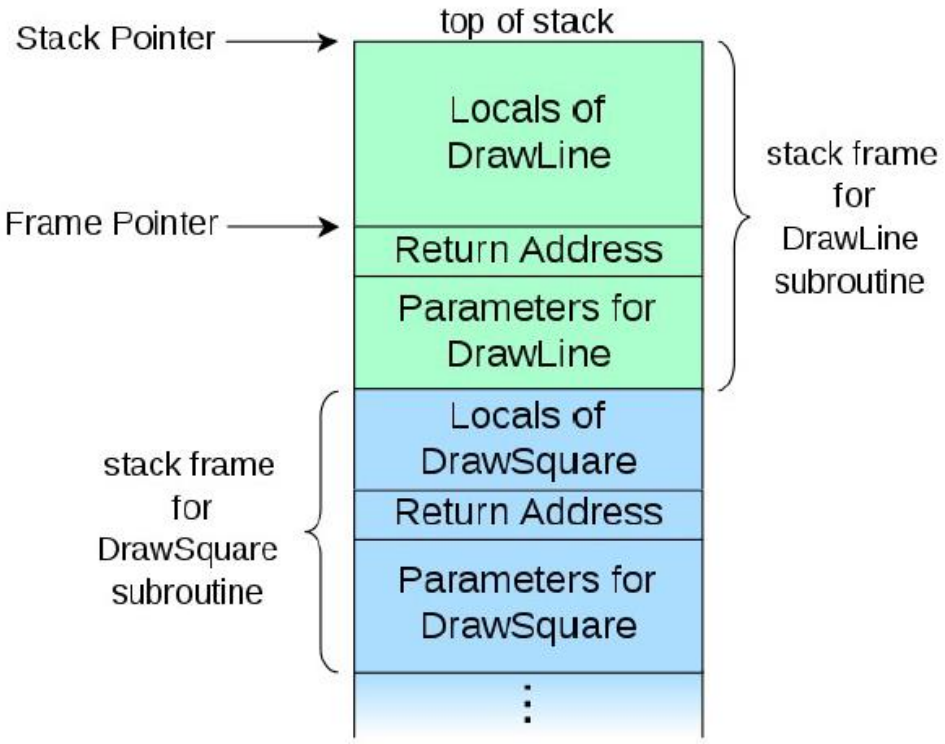


Figure 4.1: Call Stack

SAARTHAK MAHAJAN
04496403114

Return-oriented programming is an advanced version of a stack smashing attack. Generally, these types of attacks arise when an adversary manipulates the call stack by taking advantage of a bug in the program, often a buffer overrun. In a buffer overrun, a function that does not perform proper bounds checking before storing user-provided data into memory will accept more input data than it can store properly. If the data is being written onto the stack, the excess data may overflow the space allocated to the function's variables (e.g., "locals" in the stack diagram to the right) and overwrite the return address. This address will later be used by the function to redirect control flow back to the caller. If it has been overwritten, control flow will be diverted to the location specified by the new return address.

In a standard buffer overrun attack, the attacker would simply write attack code (the "payload") onto the stack and then overwrite the return address with the location of these newly written instructions. Until the late 1990s, major operating systems did not offer any protection against these attacks; Microsoft Windows provided no buffer-overrun protections until 2004. Eventually, operating systems began to combat the exploitation of buffer overflow bugs by marking the memory where data is written as non-executable, a technique known as data execution prevention. With data execution prevention enabled, the machine would refuse to execute any code located in user-writable areas of memory, preventing the attacker from placing payload on the stack and jumping to it via a return address overwrite. Hardware support for data execution prevention later became available to strengthen this protection.

With data execution prevention, an adversary cannot execute maliciously injected instructions because a typical buffer overflow overwrites contents in the data section of memory, which is marked as non-executable. To defeat this, a return-oriented programming attack does not inject malicious code, but rather uses instructions that are already present, called "gadgets", by manipulating return addresses. A typical data execution prevention cannot defend against this attack because the adversary did not use malicious code but rather combined "good" instructions by changing return addresses; therefore the code used would not be marked non-executable.

Return-into-library technique

The widespread implementation of data execution prevention made traditional buffer overflow vulnerabilities difficult or impossible to exploit in the manner described above. Instead, an attacker was restricted to code already in memory marked executable, such as the program code itself and any linked shared libraries. Since shared libraries, such as libc, often contain subroutines for performing system calls and other functionality potentially useful to an attacker, they are the most likely

SAARTHAK MAHAJAN
04496403114

candidates for finding code to assemble an attack.In a return-into-library attack, an attacker hijacks program control flow by exploiting a buffer overrun vulnerability, exactly as discussed above. Instead of attempting to write an attack payload onto the stack, the attacker instead chooses an available library function and overwrites the return address with its entry location. Further stack locations are then overwritten, obeying applicable calling conventions, to carefully pass the proper parameters to the function so it performs functionality useful to the attacker. This technique was first presented by Solar Designer in 1997 and was later extended to unlimited chaining of function calls.

Borrowed code chunks

The rise of 64-bit x86 processors brought with it a change to the subroutine calling convention that required the first argument to a function to be passed in registers instead of on the stack. This meant that an attacker could no longer set up a library function call with desired arguments just by manipulating the call stack via a buffer overrun exploit. Shared library developers also began to remove or restrict library functions that performed functions particularly useful to an attacker, such as system call wrappers. As a result, return-into-library attacks became much more difficult to successfully mount.

The next evolution came in the form of an attack that used chunks of library functions, instead of entire functions themselves, to exploit buffer overrun vulnerabilities on machines with defences against simpler attacks. This technique looks for functions that contain instruction sequences that pop values from the stack into registers. Careful selection of these code sequences allows an attacker to put suitable values into the proper registers to perform a function call under the new calling convention. The rest of the attack proceeds as a return-into-library attack.

Attacks

Return-oriented programming builds on the borrowed code chunks approach and extends it to provide Turing complete functionality to the attacker, including loops and conditional branches. Put another way, return-oriented programming provides a fully functional "language" that an attacker can use to make a compromised machine perform any operation desired. Hovav Shacham published the technique in 2007 and demonstrated how all the important programming constructs can be simulated using return-oriented programming against a target application linked with the C standard library and containing exploitable buffer overrun vulnerability.

A return-oriented programming attack is superior to the other attack types discussed both in expressive power and in resistance to defensive measures. None of the counter-exploitation techniques mentioned above, including removing potentially dangerous functions from shared libraries altogether, are effective against a return-oriented programming attack.

SAARTHAK MAHAJAN
04496403114

on x86 architecture

Although return-oriented programming attacks can be performed on a variety of architectures, Shacham's paper and a majority of follow up work focuses on the Intel x86 architecture. The x86 architecture is a variable-length CISC instruction set. Return-oriented programming on the x86 takes advantage of the fact that the instruction set is very "dense", that is, any random sequence of bytes is likely to be interpretable as some valid set of x86 instructions.

It is therefore possible to search for an opcode that alters control flow, most notably the return instruction (0xC3) and then look backwards in the binary for preceding bytes that form possibly useful instructions. These sets of instruction "gadgets" can then be chained by overwriting the return address, via a buffer overrun exploit, with the address of the first instruction of the first gadget. The first address of subsequent gadgets is then written successively onto the stack. At the conclusion of the first gadget, a return instruction will be executed, which will pop the address of the next gadget off the stack and jump to it. At the conclusion of that gadget, the chain continues with the third, and so on. By chaining the small instruction sequences, an attacker is able to produce arbitrary program behavior from pre-existing library code. Shacham asserts that given any sufficiently large quantity of code (including, but not limited to, the C standard library), sufficient gadgets will exist for Turing-complete functionality.

An automated tool has been developed to help automate the process of locating gadgets and constructing an attack against a binary. This tool, known as ROPgadget, searches through a binary looking for potentially useful gadgets, and attempts to assemble them into an attack payload that spawns a shell to accept arbitrary commands from the attacker.

on ASLR

The address space layout randomization also has vulnerabilities. According to the paper of Shacham et al, the ASLR on 32-bitarchitectures is limited by the number of bits available for address randomization. Only 16 of the 32 address bits are available for randomization, and 16 bits of address randomization can be defeated by brute force attack in minutes. For 64-bit architectures, 40 bits of 64are available for randomization. In 2016, brute force attack for 40-bits randomization are possible, but it is unlikely to go unnoticed. Also, randomization can be defeated by de-randomization technique.

Also, even with perfect randomization, leakage of memory contents will help to calculate the base address of a DLL at runtime.

SAARTHAK MAHAJAN
04496403114

No use of return instruction

According to the paper of Checkoway et al, it is possible to perform return-oriented-programming on X86 and ARM architectures without using return instruction: 0xC3. They instead used carefully crafted instruction sequences that already exist in the machine to behave like return instruction. Return instruction has 2 effects:

(1) The instruction searches for the four-byte value at the top of the stack, and set the instruction pointer to the value.

(2) It increases the stack pointer value by four. On X86 architecture, sequences of jmp and pop instructions can act as a return instruction. On ARM, sequences of load and branch instructions can act as a return instruction.

Since this new approach does not use return instruction, it has negative implications for defense. When defense program check not only for several returns but also for several jump instructions, this attack will be detected.

Defences

Unbroken defense

The G-free technique was developed by Kaan Onarlioglu, Leyla Bilge, Andrea Lanzi, Davide Balzarotti, and Engin Kirda. It is a practical solution against any possible form of return-oriented programming. The solution eliminates all unaligned free-branch instructions (instructions like RET or CALL which attackers can use to change control flow) inside a binary executable, and protects the free-branch instructions from being used by an attacker. The way G-free protects the return address is similar to the XOR canary implemented by StackGuard. Also, it checks authenticity of function calls by appending validation block. If the decreed result is not found or does not match, then G-free causes the application to crash.

Address Space Layout Randomization

A number of techniques have been proposed to subvert attacks based on return-oriented programming. Most rely on randomizing the location of program and library code, so that an attacker cannot accurately predict the location of instructions that might be useful in gadgets and therefore cannot mount a successful return-oriented programming attack chain. One fairly common implementation of this technique, address space layout randomization (ASLR), loads shared libraries into a different memory location at each program load.

Although widely deployed by modern operating systems, ASLR is vulnerable to information leakage attacks and other approaches to determine the address of any known library function in memory. If an attacker can successfully determine the

location of one known instruction, the position of all others can be inferred and a return-oriented programming attack can be constructed.

This randomization approach can be taken further by relocating all the instructions and/or other program state (registers and stacks objects) of the program separately, instead of just library locations. This requires extensive runtime support, such as a software dynamic translator, to piece the randomized instructions back together at runtime. This technique is successful at making gadgets difficult to find and utilize, but comes with significant overhead.

Another approach, taken by kBouncer, modifies the operating system to verify that return instructions actually divert control flow back to a location immediately following a call instruction. This prevents gadget chaining, but carries a heavy performance penalty and is not effective against jump-oriented programming attacks which alter jumps and other control-flow-modifying instructions instead of returns.

DEP/NX

Data Execution Prevention/No-execute is a defense application developed by Microsoft. DEP/NX alone does not provide perfect protection against all memory-based vulnerabilities, but as with ASLR technologies, it can provide advanced protection on memory based vulnerabilities.

SEHOP

Structured Exception Handler Overwrite Protection is a feature of Windows which provide a protection against the most common stack overflow attacks, especially against attacks on structured exception handler.

Against Return-Oriented Rootkits

One approach for defending return-oriented programming would be to create a compiler-based defense mechanism that eliminates return instructions so that an adversary or return-oriented rootkits cannot make return-oriented gadgets. To eliminate return instructions, a technique called return indirection replaces the return address in the stack frame with an index.

SAARTHAK MAHAJAN
04496403114

CHAPTER 5

CONTROL FLOW GRAPH & CONTROL FLOW INTEGRITY

A **control flow graph** (**CFG**) in computer science is a representation, using graph notation, of all paths that might be traversed through a program during its execution. The control flow graph is due to Frances E. Allen, who notes that Reese T. Prosser used Boolean connectivity matrices for flow analysis before. The CFG is essential to many compiler optimizations and static analysis tools.

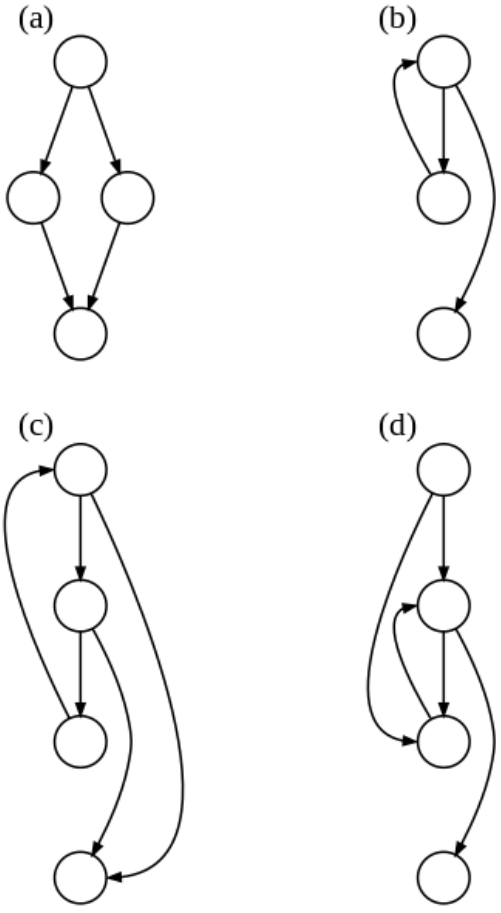


Figure 5.1: CFG examples – (a) if-then-else (b) while loop (c) a natural loop with two exits (d) an irreducible CFG – loop with two entry points

SAARTHAK MAHAJAN
04496403114

In a control flow graph each node in the graph represents a basic block, i.e. a straight-line piece of code without any jumps or jump targets; jump targets start a block, and jumps end a block. Directed edges are used to represent jumps in the control flow. There are, in most presentations, two specially designated blocks: the *entry block*, through which control enters into the flow graph, and the *exit block*, through which all control flow leaves.

Because of its construction procedure, in a CFG, every edge A→B has the property that:

Outdegree (A) > 1 or Indegree(B) > 1 (or both).

The CFG can thus be obtained, at least conceptually, by starting from the program's (full) flow graph—i.e. the graph in which every node represents an individual instruction—and performing an edge contraction for every edge that falsifies the predicate above, i.e. contracting every edge whose source has a single exit and whose destination has a single entry. This contraction-based algorithm is of no practical importance, except as a visualization aid for understanding the CFG construction, because the CFG can be more efficiently constructed directly from the program by scanning it for basic blocks.

- Control-Flow Integrity (CFI) restricts the control-flow of an application to *valid* execution traces.
- CFI enforces this property by monitoring the program at runtime and comparing its state to a set of precomputed valid states.
- If an invalid state is detected, an alert is raised, usually terminating the application.
- Any CFI mechanism consists of two abstract components : the (often static) analysis component that recovers the Control-Flow Graph (CFG) and the dynamic enforcement mechanism that restricts control flows according to the generated CFG.
- A CFG is a graph that covers all valid executions of the program. Nodes in the graph are locations of control-flow transfers in the program and edges encode reachable targets. The CFG is an abstract concept and the different existing CFI mechanisms use different approaches to generate the underlying CFGs using both static and dynamic analysis, relying on either the binary or source code.
- For forward edges, the CFG generation enumerates all possible targets, often leveraging information from the underlying source language. Switch statements in C/C++ are a good example as the different targets are statically known and the compiler can generate a fixed jump table and emit an indirect jump with a bound check to guarantee that the target used at runtime is one of the valid targets in the switch statement.

SAARTHAK MAHAJAN
04496403114

- For indirect function calls through a function pointer, the underlying analysis becomes more complicated as the target may not be known a-priori. Common source-based analyses use a type-based approach and, looking at the function prototype of the function pointer that is used, enumerate all matching functions. Different CFI mechanisms use different forms of type equality, e.g., any valid function, functions with the same arity (number of arguments), or functions with the same signature (arity and equivalence of argument types). At runtime, any function with matching signature is allowed.

- Just looking at function prototypes likely yields several collisions where functions are reachable that may never be called in practice. The analysis therefore over-approximates the valid set of targets. In practice, the compiler can check which functions are **address taken**, i.e., there is a source line that generates the address of the function and stores it. The CFI mechanism may reduce the number of allowed targets by intersecting the sets of equal function prototypes and the set of address taken functions.

- For virtual calls, i.e., indirect calls in C++ that depend on the type of the object and the class relationship, the analysis can further leverage the type of the object to restrict the valid functions, e.g., all object constructors have the same signature but only the subset constructors of related classes are feasible.

- So far, the constructed CFG is stateless, i.e., the *context of the execution* is not considered and each control-flow transfer is independent of all others. On one hand, at runtime only one target is allowed for any possible transfer, namely the target address currently stored at the memory location of the code pointer. CFG construction, on the other hand, over-approximates the number of valid targets with different granularities, depending on the precision of the analysis. Some mechanisms take path constraints into consideration and check (for a limited depth) if the path taken through the application is feasible by using a dynamic analysis approach that validates the current execution. So far, only few mechanisms look at the path context as this incurs dynamic tracking costs at runtime.

SUMMARY

Control flow analysis aims to determine the execution order of program statements or instructions.

Basic block: a linear sequence of program instructions having one entry point (the first instruction executed) and one exit point (the last instruction executed).

Control flow graph (CFG) is a directed graph in which the nodes represent basic blocks and the edges represent control flow paths.

A control flow graph specifies all possible execution path.

SAARTHAK MAHAJAN
04496403114

CHAPTER 6

CFI ENFORCEMENT (INSTRUMENTATION)

In the context of computer programming, **instrumentation** refers to an ability to monitor or measure the level of a product's performance, to diagnose errors and to write trace information. Programmers implement instrumentation in the form of code instructions that monitor specific components in a system (for example, instructions may output logging information to appear on screen). When an application contains instrumentation code, it can be managed using a management tool. Instrumentation is necessary to review the performance of the application. Instrumentation approaches can be of two types: Source instrumentation and binary instrumentation.

In programming, instrumentation means the ability of an application to incorporate:

- Code tracing - receiving informative messages about the execution of an application at run time.
- Debugging and (structured) exception handling - tracking down and fixing programming errors in an application under development.
- Profiling - a means by which dynamic program behaviors can be measured during a training run with a representative input. This is useful for properties of a program which cannot be analyzed statically with sufficient precision, such as alias analysis.
- Performance counters - components that allow the tracking of the performance of the application.
- Computer data logging - components that allow the logging and tracking of major events in the execution of the application.

Some performance measurement tools add instrumentation to the code. E.g. they may binary translate, and add instructions to read the timers at the beginning and end of functions. Or this instrumentation, this reading of the timers, may be added to assembly, or C code, by an automated tool, or a programmer.

Other performance measurement tools do not change the code that is being measured. E.g. UNIX prof sampling runs special code that is invoked at the timer interrupt, which generates a histogram of the instruction at which the interrupt is received.

SAARTHAK MAHAJAN
04496403114

Some tools are hybrid: e.g. UNIX gprof combines prof-style interrupt sampling with mcount instrumentation added by the compiler with the -pg option to count which functions call each other.

All performance measurement has overhead, but instrumentation tends to have more overhead than interrupt based sampling. On the other hand, instrumentation can measure more stuff.

Static Instrumentation

In this approach the instrumentation occurs before your program even runs. Essentially the probes are baked in to you binary. There are two ways this can be done. One is to have the compiler put the probes in. Visual C++ takes this approach to implant Profile Guided Optimizations (PGO). Another is to do what is called *post link* instrumentation. With this approach you take the binary that has been emitted by the compiler, crack it open to find all the methods, instrument and then rewrite the binary like nothing ever happened.

PROS AND CONS

This approach yields much faster start-up times. The images that is loaded in to memory is the image that runs.

The rewrite process destroys .NET strong naming, and thus images must be resigned after instrumentation, or verification skipping must be enabled. Statically instrumented images can also be placed in the *native image cache.*

Rewriting the images means you have to deal with the whole question of where to put the files that you instrument. This can be quite a housekeeping chore for the user.

Rewriting PE files correctly is relatively straightforward for .NET applications. It's really hard for native applications, particularly for x64 images, where instrumenting methods can turn them from leaf methods to non-leaf methods.

SAARTHAK MAHAJAN
04496403114

Dynamic Instrumentation.

In this approach the instrumentation occurs as the program is running. In the case of a just in time compilation environment such as .NET, this is achieved by using the .NET profiling API to get notified when methods are about to be compiled and executed for the first time. Your profiler gets the opportunity to rewrite the method that is about to be run, by adding probes or whatever you want. Then the program continues on. Dynamic instrumentation can also be performed on native images too, but doing this is hard and rarely done. The approaches you could take to pull this involve injecting a DLL into the process, suspending it and then using debug information to patch and rewrite methods.

PROS AND CONS

Yields significantly slower start-up times and increased memory usage because each method body must be reallocated, as well as the some CPU load to instrument on the fly.

If you want to do instrumentation at the line level, this approach requires loading additional data from the .pdb files in at runtime too. If you load the actual .pdb's that has a huge runtime cost.

Because the original file was not modified, .NET strong naming is not broken. However, profiling of images in the *native image cache* must be disabled, as these cannot be dynamically instrumented because they are already compiled to native code.

Differences

This highlights one of the key differences between static binary analysis and dynamic binary analysis. Rather than considering what may occur, dynamic binary analysis has the benefit of operating on what actually does occur.

CHAPTER 7

INTEL PIN TOOL

Figure 7.1: Pintool

7.1 INTRODUCTION

Pin is a platform for creating analysis tools. A pin tool comprises instrumentation, analysis and callback routines. Instrumentation routines are called when code that has not yet been recompiled is about to be run, and enable the insertion of analysis routines. Analysis routines are called when the code associated with them is run. Callback routines are only called when specific conditions are met, or when a certain event has occurred. Pin provides an extensive application programming interface (API) for instrumentation at different abstraction levels, from one instruction to an entire binary module. It also supports callbacks for many events such as library loads, system calls, signals/exceptions and thread creation events.

Pin performs instrumentation by taking control of the program just after it loads into the memory. Then just-in-time recompiles (JIT) small sections of the binary code using pin just before it is run. New instructions to perform analysis are added to the recompiled code. These new instructions come from the Pintool. A large array of optimization techniques are used to obtain the lowest possible running time and memory use overhead. As of June 2010, Pin's average base overhead is 30 percent (without running a Pintool).

SAARTHAK MAHAJAN
04496403114

Features

Instrumentation modes

Pin supports two modes of instrumentation called JIT mode and Probe mode. JIT mode supports all features of Pin, while Probe mode supports a limited feature set but is far faster, adding almost no overhead to program's running time. JIT mode uses a just-in-time compiler to recompile all program code and insert instrumentation, while Probe mode uses code trampolines for instrumentation.

Platform independence

Pin was designed for tool portability, and despite JIT compiling from one ISA to the same ISA (and not using a single intermediate representation for all code), most of its APIs are architecture and operating system independent. It was also designed to be portable itself, carefully isolating platform-specific code from generic code, allowing the fast adaptation of Pin to new platforms. Approximately half of the code is generic and the rest is either architecture or OS dependent.

Optimizations

Pin uses many techniques to optimize instrumentation and analysis code, using techniques such as inlining, liveness analysis and smart register spilling. Pin performs these optimizations automatically whenever possible, without needing users to insert any extra code to allow inlining. Naturally, some optimizations still require user hints, and some code structures are easier to inline than others. Direct linking of jitted code sections, a technique called *trace linking*, and *register binding reconciliation*, which minimizes register spilling and remapping, are also used.

Ease of use

Pin's API and implementation are focused on making pin tools easy to write. Pin takes full responsibility for assuring that the instrumentation code from the pin tool does not affect the application state. Also, the API enables instrumentation code to request many pieces of information from Pin. For example, the instrumentation code in the pin tool can use the Pin API to get the memory address being accessed by an instruction, without having to examine the instruction in detail.

SAARTHAK MAHAJAN
04496403114

Uses

Uses as a Defence Mechanism

Some scholars think that one can use Pin tool or binary instrumentation techniques to detect malware. Unlike traditional antiviruses where scanning files is used to detect viruses, one can use tools like Pin tool to scan program's resources to detect abnormalities; thus detect malware.

Utilizing System Resource Monitoring

Unlike traditional techniques of scanning files, this approach doesn't need to be updated regularly and uses a more efficient way to detect malwares rather than burdening the processor by scanning all the files. This approach keeps track of the system's resources used by a program and terminates the program if its resource usage goes beyond a given threshold limit.

Alternatives to Pin Tool

DynamoRIO is a process virtual machine that redirects a program's execution from its original binary code to a copy of that code. Instrumentation that carries out the actions of the desired tool, are then added to this copy. No changes are made to the original program, which does not need to be specially prepared in any way. DynamoRIO operates completely at run time and handles legacy code, dynamically loaded libraries, dynamically generated code, and self-modifying code. DynamoRIO monitors all control flow to capture the entire execution of the target program. This monitoring adds overhead even when no tool is present. DynamoRIO's average overhead is 11 percent.

Valgrind is in essence a virtual machine using just-in-time (JIT) compilation techniques, including dynamic recompilation. Nothing from the original program ever gets run directly on the host processor. Instead, Valgrind first translates the program into a temporary, simpler form called Intermediate Representation (IR), which is a processor-neutral, SSA-based form. After the conversion, a *tool* is free to do whatever transformations it would like on the IR, before Valgrind translates the IR back into machine code and lets the host processor run it. Valgrind recompiles binary code to run on host and target (or simulated) CPUs of the same architecture. It also includes a GDB stub to allow debugging of the target program as it runs in Valgrind, with "monitor commands" that allow you to query the Valgrind tool for various sorts of information. A considerable amount of performance is lost in these transformations (and usually, the code the tool inserts); usually, code run with Valgrind and the "none" tool (which does nothing to the IR) runs at 1/4 to 1/5 of the speed of the normal program.

7.2 HOW TO INSTRUMENT WITH PIN

Pin

The best way to think about Pin is as a "just in time" (JIT) compiler. The input to this compiler is not bytecode, however, but a regular executable. Pin intercepts the execution of the first instruction of the executable and generates ("compiles") new code for the straight line code sequence starting at this instruction. It then transfers control to the generated sequence. The generated code sequence is almost identical to the original one, but Pin ensures that it regains control when a branch exits the sequence. After regaining control, Pin generates more code for the branch target and continues execution. Pin makes this efficient by keeping all of the generated code in memory so it can be reused and directly branching from one sequence to another.

In JIT mode, the only code ever executed is the generated code. The original code is only used for reference. When generating code, Pin gives the user an opportunity to inject their own code (instrumentation).

Pin instruments all instructions that are actually executed. It does not matter in what section they reside. Although there are some exceptions for conditional branches, generally speaking, if an instruction is never executed then it will not be instrumented.

Pintools

Conceptually, instrumentation consists of two components:

- A mechanism that decides where and what code is inserted
- The code to execute at insertion points

These two components are *instrumentation* and *analysis* code. Both components live in a single executable, a *Pintool*. Pintools can be thought of as plugins that can modify the code generation process inside Pin.

The Pintool registers instrumentation callback routines with Pin that are called from Pin whenever new code needs to be generated. This instrumentation callback routine represents the instrumentation component. It inspects the code to be generated, investigates its static properties, and decides if and where to inject calls to analysis functions.

The analysis function gathers data about the application. Pin makes sure that the integer and floating point register state is saved and restored as necessary and allow arguments to be passed to the functions.

SAARTHAK MAHAJAN
04496403114

The Pintool can also register notification callback routines for events such as thread creation or forking. These callbacks are generally used to gather data or tool initialization or clean up.

Observations

Since a Pintool works like a plugin, it must run in the same address space as Pin and the executable to be instrumented. Hence the Pintool has access to all of the executable's data. It also shares file descriptors and other process information with the executable.

Pin and the Pintool control a program starting with the very first instruction. For executables compiled with shared libraries this implies that the execution of the dynamic loader and all shared libraries will be visible to the Pintool.

When writing tools, it is more important to tune the analysis code than the instrumentation code. This is because the instrumentation is executed once, but analysis code is called many times.

Instrumentation Granularity

As described above, Pin's instrumentation is "just in time" (JIT). Instrumentation occurs immediately before a code sequence is executed for the first time. We call this mode of operation *trace instrumentation.*

Trace instrumentation lets the Pintool inspect and instrument an executable one trace at a time. Traces usually begin at the target of a taken branch and end with an unconditional branch, including calls and returns. Pin guarantees that a trace is only entered at the top, but it may contain multiple exits. If a branch joins the middle of a trace, Pin constructs a new trace that begins with the branch target. Pin breaks the trace into basic blocks, *BBLs*. A BBL is a single entrance, single exit sequence of instructions. Branches to the middle of a bbl begin a new trace and hence a new BBL. It is often possible to insert a single analysis call for a BBL, instead of one analysis call for every instruction. Reducing the number of analysis calls makes instrumentation more efficient. Trace instrumentation utilizes the **TRACE_AddInstrumentFunction** API call.

Note, though, that since Pin is discovering the control flow of the program dynamically as it executes, Pin's BBL can be different from the classical definition of a BBL which you will find in a compiler textbook.

SAARTHAK MAHAJAN
04496403114

Pin also breaks BBLs on some other instructions which may be unexpected, for instance cpuid, popf and REP prefixed instructions all end traces and therefore BBLs. Since REP prefixed instructions are treated as implicit loops, if a REP prefixed instruction iterates more than once, iterations after the first will cause a single instruction BBL to be generated, so in this case you would see more basic blocks executed than you might expect.

As a convenience for Pintool writers, Pin also offers an *instruction instrumentation* mode which lets the tool inspect and instrument an executable a single instruction at a time. This is essentially identical to trace instrumentation where the Pintool writer has been freed from the responsibility of iterating over the instructions inside a trace. As described under trace instrumentation, certain BBLs and the instructions inside of them may be generated (and hence instrumented) multiple times. Instruction instrumentation utilizes the **INS_AddInstrumentFunction** API call.

Sometimes, however, it can be useful to look at different granularity than a trace. For this purpose Pin offers two additional modes: image and routine instrumentation. These modes are implemented by "caching" instrumentation requests and hence incur a space overhead, these modes are also referred to as ahead-of-time instrumentation.

Image instrumentation lets the Pintool inspect and instrument an entire image, IMG, when it is first loaded. A Pintool can walk the sections, SEC, of the image, the routines, RTN, of a section, and the instructions, INS of a routine. Instrumentation can be inserted so that it is executed before or after a routine is executed, or before or after an instruction is executed. Image instrumentation utilizes the **IMG_AddInstrumentFunction** API call. Image instrumentation depends on symbol information to determine routine boundaries hence **PIN_InitSymbols** must be called before **PIN_Init**.

Routine instrumentation lets the Pintool inspect and instrument an entire routine when the image it is contained in is first loaded. A Pintool can walk the instructions of a routine. There is not enough information available to break the instructions into BBLs. Instrumentation can be inserted so that it is executed before or after a routine is executed, or before or after an instruction is executed. Routine instrumentation is provided as a convenience for Pintool writers, as an alternative to walking the sections and routines of the image during the Image instrumentation, as described in the previous paragraph.

Routine instrumentation utilizes the **RTN_AddInstrumentFunction** API call. Instrumentation of routine exits does not work reliably in the presence of tail calls or when return instructions cannot reliably be detected.

SAARTHAK MAHAJAN
04496403114

Instrumenting Multi-threaded Applications

Instrumenting a multi-threaded program requires that the tool be thread safe - access to global storage must be coordinated with other threads. Pin tries to provide a conventional C++ program environment for tools, but it is not possible to use the standard library interfaces to manage threads in a Pintool. For example, Linux tools cannot use the pthreads library and Windows tools should not use the Win32 API's to manage threads. Instead, Pin provides its own locking and thread management API's, which the Pintool should use. (See **LOCK: Locking Primitives** and **Pin Thread API**.)

Pintools do not need to add explicit locking to instrumentation routines because Pin calls these routines while holding an internal lock called the VM lock. However, Pin does execute analysis and replacement functions in parallel, so Pintools may need to add locking to these routines if they access global data.

Pintools on Linux also need to take care when calling standard C or C++ library routines from analysis or replacement functions because the C and C++ libraries linked into Pintools are **not** thread-safe. Some simple C / C++ routines are safe to call without locking, because their implementations are inherently thread-safe, however, Pin does not attempt to provide a list of safe routines. If you are in doubt, you should add locking around calls to library functions. In particular, the "errno" value is not multi-thread safe, so tools that use this should provide their own locking. Note that these restrictions only exist on the Unix platforms, as the library routines on Windows are thread safe.

Pin provides call-backs when each thread starts and ends (see PIN_AddThreadStartFunction and PIN_AddThreadFiniFunction). These provide a convenient place for a Pintool to allocate and manipulate thread local data and store it on a thread's local storage.

Pin also provides an analysis routine argument (IARG_THREAD_ID), which passes a Pin-specific thread ID for the calling thread. This ID is different from the O/S system thread ID, and is a small number starting at 0, which can be used as an index to an array of thread data or as the locking value to Pin user locks. See the example **Instrumenting Threaded Applications** for more information.

In addition to the Pin thread ID, the Pin API provides an efficient thread local storage (TLS), with the option to allocate a new TLS key and associate it with a given data destruction function. Any thread of the process can store and retrieve values in its own slot, referenced by the allocated key. The initial value associated with the key in all threads is NULL. See the example **Using TLS** for more information.

SAARTHAK MAHAJAN
04496403114

False sharing occurs when multiple threads access different parts of the same cache line and at least one of them is a write. To maintain memory coherency, the computer must copy the memory from one CPU's cache to another, even though data is not truly shared. False sharing can usually be avoided by padding critical data structures to the size of a cache line, or by rearranging the data layout of structures. See the example **Using TLS** for more information.

Avoiding Deadlocks in Multi-threaded Applications

Since Pin, the tool, and the application may each acquire and release locks, Pin tool developers must take care to avoid deadlocks with either the application or Pin. Deadlocks generally occur when two threads acquire the same locks in a different order. For example, thread A acquires lock L1 and then acquires lock L2, while thread B acquires lock L2 and then acquires lock L1. This will lead to a deadlock if thread A holds lock L1 and waits for L2 while thread B holds lock L2 and waits for L1. To avoid such deadlocks, Pin imposes a hierarchy on the order in which locks must be acquired. Pin generally acquires its own internal locks before the tool acquires any lock (e.g. via **PIN_GetLock()**). Additionally, we assume that the application may acquire locks at the top of this hierarchy (i.e. before Pin acquires its internal locks). The following diagram illustrates the hierarchy:

Application locks -> Pin internal locks -> Tool locks

Pin tool developers should design their Pin tools such that they never break this lock hierarchy, and they can do so by following these basic guidelines:

- If the tool acquires any locks from within a Pin call-back, it must release those locks before returning from that call-back. Holding a lock across Pin call-backs violates the hierarchy with respect to the Pin internal locks.
- If the tool acquires any locks from within an analysis routine, it must release those locks before returning from the analysis routine. Holding a lock across Pin analysis routines violates the hierarchy with respect to Pin internal locks and other locks used by the instrumented application itself.
- If the tool calls a Pin API from within a Pin call-back or analysis routine, it should not hold any tool locks when calling the API. Some of the Pin APIs use the internal Pin locks so holding a tool lock before invoking these APIs violates the hierarchy with respect to the Pin internal locks.
- If the tool calls a Pin API from within an analysis routine, it may need to acquire the Pin client lock first by calling **PIN_LockClient()**. This depends on the API, so check the documentation for the specific API for more information. Note that the tool should not hold any other locks when calling **PIN_LockClient()**, as described in the previous item.

SAARTHAK MAHAJAN
04496403114

While these guidelines are sufficient in most cases, they may turn out to be too restrictive for certain use-cases. The next set of guidelines explains the conditions in which it is safe to relax the basic guidelines above:

- In JIT mode, the tool may acquire locks from within an analysis routine and not release them, providing it releases these locks before leaving the trace that contains the analysis routine. The tool must expect that the trace may exit "early" if an application instruction raises an exception. Any lock L, which the tool might hold when the application raises an exception, must obey the following sub-rules:
    - The tool must establish a call-back that executes when the application raises an exception and this call-back must release lock L if it was acquired at the time the exception occurred. Tools can use **PIN_AddContextChangeFunction()** to establish this call-back.
    - The tool must not acquire lock L from within any Pin call-back, to avoid violating the hierarchy with respect to the Pin internal locks.

- If the tool calls a Pin API from an analysis routine, it may acquire and hold a lock L while calling the API providing that:
    - Lock L is not being acquired from any Pin call-back. This avoids the hierarchy violation with respect to the Pin internal locks.
    - The Pin API being invoked does not cause application code to execute (e.g., **PIN_CallApplicationFunction()**). This avoids the hierarchy violation with respect to the locks used by the application itself.
- Pin is a tool for the instrumentation of programs. It supports the Android*, Linux*, OS X* and Windows* operating systems and executables for the IA-32, Intel(R) 64 and Intel(R) Many Integrated Core architectures.
- Pin allows a tool to insert arbitrary code (written in C or C++) in arbitrary places in the executable. The code is added dynamically while the executable is running. This also makes it possible to attach Pin to an already running process.
- Pin provides a rich API that abstracts away the underlying instruction set idiosyncrasies and allows context information such as register contents to be passed to the injected code as parameters. Pin automatically saves and restores the registers that are overwritten by the injected code so the application continues to work. Limited access to symbol and debug information is available as well.
- Pin includes the source code for a large number of example instrumentation tools like basic block profilers, cache simulators, instruction trace generators, etc. It is easy to derive new tools using the examples as a template.

7.3 Examples

Building the Example Tools

- To build all examples in a directory for ia32 architecture:

  $ cd source/tools/ManualExamples

  $ make all TARGET=ia32

- To build all examples in a directory for intel64 architecture:

  $ cd source/tools/ManualExamples

  $ make all TARGET=intel64

- To build and run a specific example (e.g., inscount0):

  $ cd source/tools/ManualExamples

  $ make inscount0.test TARGET=intel64

- To build a specific example without running it (e.g., inscount0):

  $ cd source/tools/ManualExamples

  $ make obj-intel64/inscount0.so TARGET=intel64

The above applies to the Intel(R) 64 architecture. For the IA-32 architecture, use TARGET=ia32 instead.

$ cd source/tools/ManualExamples

$ make obj-ia32/inscount0.so TARGET=ia32

Notes for Building Tools for Windows

Since the tools are built using make, be sure to install cygwin make first.

Open the Visual Studio Command Prompt corresponding to your target architecture, i.e. x86 or x64, and follow the steps in the **Building the Example Tools** section.

SAARTHAK MAHAJAN
04496403114

Simple Instruction Count (Instruction Instrumentation)

The example below instruments a program to count the total number of instructions executed. It inserts a call to docount before every instruction. When the program exits, it saves the count in the file inscount.out.

Here is how to run it and display its output (note that the file list is the ls output, so it may be different on your machine, similarly the instruction count will depend on the implementation of ls):

```
$ ../../../pin -t obj-intel64/inscount0.so -- /bin/ls
Makefile          atrace.o     imageload.out  itrace      proccount
Makefile.example  imageload    inscount0      itrace.o    proccount.o
atrace            imageload.o  inscount0.o    itrace.out
$ cat inscount.out
Count 422838
$
```

Figure 7.2: Output for Instruction Count

The example can be found in source/tools/ManualExamples/inscount0.cpp

```
#include <iostream>
#include <fstream>
#include "pin.H"

ofstream OutFile;

// The running count of instructions is kept here
// make it static to help the compiler optimize docount
static UINT64 icount = 0;

// This function is called before every instruction is executed
VOID docount() { icount++; }

// Pin calls this function every time a new instruction is encountered
VOID Instruction(INS ins, VOID *v)
{
    // Insert a call to docount before every instruction, no arguments are passed
    INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)docount, IARG_END);
}

KNOB<string> KnobOutputFile(KNOB_MODE_WRITEONCE, "pintool",
    "o", "inscount.out", "specify output file name");

// This function is called when the application exits
VOID Fini(INT32 code, VOID *v)
{
    // Write to a file since cout and cerr maybe closed by the application
    OutFile.setf(ios::showbase);
    OutFile << "Count " << icount << endl;
    OutFile.close();
}

/* ===================================================================== */
/* Print Help Message                                                    */
/* ===================================================================== */

INT32 Usage()
{
    cerr << "This tool counts the number of dynamic instructions executed" << endl;
    cerr << endl << KNOB_BASE::StringKnobSummary() << endl;
    return -1;
}
```


SAARTHAK MAHAJAN
04496403114

```
/* ===================================================================== */
/* Main                                                                  */
/* ===================================================================== */
/*   argc, argv are the entire command line: pin -t <toolname> -- ...    */
/* ===================================================================== */

int main(int argc, char * argv[])
{
    // Initialize pin
    if (PIN_Init(argc, argv)) return Usage();

    OutFile.open(KnobOutputFile.Value().c_str());

    // Register Instruction to be called to instrument instructions
    INS_AddInstrumentFunction(Instruction, 0);

    // Register Fini to be called when the application exits
    PIN_AddFiniFunction(Fini, 0);

    // Start the program, never returns
    PIN_StartProgram();

    return 0;
}
```

Instruction Address Trace (Instruction Instrumentation)

In the previous example, we did not pass any arguments to docount, the analysis procedure. In this example, we show how to pass arguments. When calling an analysis procedure, Pin allows you to pass the instruction pointer, current value of registers, effective address of memory operations, constants, etc. For a complete list, see **IARG_TYPE**.

With a small change, we can turn the instruction counting example into a Pintool that prints the address of every instruction that is executed. This tool is useful for understanding the control flow of a program for debugging, or in processor design when simulating an instruction cache.

We change the arguments to INS_InsertCall to pass the address of the instruction about to be executed. We replace docount with printip, which prints the instruction address. It writes its output to the file itrace.out.

This is how to run it and look at the output:

SAARTHAK MAHAJAN
04496403114

```
$ ../../../pin -t obj-intel64/itrace.so -- /bin/ls
Makefile          atrace.o     imageload.out  itrace      proccount
Makefile.example  imageload    inscount0      itrace.o    proccount.o
atrace            imageload.o  inscount0.o    itrace.out
$ head itrace.out
0x40001e90
0x40001e91
0x40001ee4
0x40001ee5
0x40001ee7
0x40001ee8
0x40001ee9
0x40001eea
0x40001ef0
0x40001ee0
$
```

Figure 7.3: Output for Trace Instrumentation

The example can be found in source/tools/ManualExamples/itrace.cpp

```
#include <stdio.h>
#include "pin.H"

FILE * trace;

// This function is called before every instruction is executed
// and prints the IP
VOID printip(VOID *ip) { fprintf(trace, "%p\n", ip); }

// Pin calls this function every time a new instruction is encountered
VOID Instruction(INS ins, VOID *v)
{
    // Insert a call to printip before every instruction, and pass it the IP
    INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)printip, IARG_INST_PTR, IARG_END);
}

// This function is called when the application exits
VOID Fini(INT32 code, VOID *v)
{
    fprintf(trace, "#eof\n");
    fclose(trace);
}

/* ===================================================================== */
/* Print Help Message                                                    */
/* ===================================================================== */

INT32 Usage()
{
    PIN_ERROR("This Pintool prints the IPs of every instruction executed\n"
              + KNOB_BASE::StringKnobSummary() + "\n");
    return -1;
}
```


SAARTHAK MAHAJAN
04496403114

```
/* ===================================================================== */
/* Main                                                                  */
/* ===================================================================== */

int main(int argc, char * argv[])
{
    trace = fopen("itrace.out", "w");

    // Initialize pin
    if (PIN_Init(argc, argv)) return Usage();

    // Register Instruction to be called to instrument instructions
    INS_AddInstrumentFunction(Instruction, 0);

    // Register Fini to be called when the application exits
    PIN_AddFiniFunction(Fini, 0);

    // Start the program, never returns
    PIN_StartProgram();

    return 0;
}
```

Memory Reference Trace (Instruction Instrumentation)

The previous example instruments all instructions. Sometimes a tool may only want to instrument a class of instructions, like memory operations or branch instructions. A tool can do this by using the Pin API which includes functions that classify and examine instructions. The basic API is common to all instruction sets and is described **here.** In addition, there is an instruction set specific API for the **IA-32 ISA.**

In this example, we show how to do more selective instrumentation by examining the instructions. This tool generates a trace of all memory addresses referenced by a program. This is also useful for debugging and for simulating a data cache in a processor.

We only instrument instructions that read or write memory. We also use **INS_InsertPredicatedCall** instead of **INS_InsertCall** to avoid generating references to instructions that are predicated when the predicate is false. On IA-32 and Intel(R) 64 architectures CMOVcc, FCMOVcc and REP prefixed string operations are treated as being predicated. For CMOVcc and FCMOVcc the predicate is the condition test implied by "cc", for REP prefixed string ops it is that the count register is non-zero.

Since the instrumentation functions are only called once and the analysis functions are called every time an instruction is executed, it is much faster to instrument only the memory operations, as compared to the previous instruction trace example that instruments every instruction.

SAARTHAK MAHAJAN
04496403114

Here is how to run it and the sample output:

```
$ ../../../pin -t obj-intel64/pinatrace.so -- /bin/ls
Makefile          atrace.o    imageload.o    inscount0.o  itrace.out
Makefile.example  atrace.out  imageload.out  itrace       proccount
atrace            imageload   inscount0      itrace.o     proccount.o
$ head pinatrace.out
0x40001ee0: R 0xbfffe798
0x40001efd: W 0xbfffe7d4
0x40001f09: W 0xbfffe7d8
0x40001f20: W 0xbfffe864
0x40001f20: W 0xbfffe868
0x40001f20: W 0xbfffe86c
0x40001f20: W 0xbfffe870
0x40001f20: W 0xbfffe874
0x40001f20: W 0xbfffe878
0x40001f20: W 0xbfffe87c
$
```

Figure 7.4: Output for Memory Trace Instrumentation

Detecting the Loading and Unloading of Images (Image Instrumentation)

The example below prints a message to a trace file every time and image is loaded or unloaded. It really abuses the image instrumentation mode as the Pintool neither inspects the image nor adds instrumentation code.

If you invoke it on ls, you would see this output:

```
$ ../../../pin -t obj-intel64/imageload.so -- /bin/ls
Makefile          atrace.o    imageload.o    inscount0.o  proccount
Makefile.example  atrace.out  imageload.out  itrace       proccount.o
atrace            imageload   inscount0      itrace.o     trace.out
$ cat imageload.out
Loading /bin/ls
Loading /lib/ld-linux.so.2
Loading /lib/libtermcap.so.2
Loading /lib/i686/libc.so.6
Unloading /bin/ls
Unloading /lib/ld-linux.so.2
Unloading /lib/libtermcap.so.2
Unloading /lib/i686/libc.so.6
$
```

Figure 7.5: Output for Image Instrumentation

SAARTHAK MAHAJAN
04496403114

7.4 BACKWARD EDGE INSTRUMENTATION

INSTRUMENTATION OF BACKWARD EDGE TRANSFERS (SHADOW STACK)

- **Shadow Stack** is a mechanism for maintaining control-flow integrity by mitigating return address overwrites such as those seen during exploitation of a stack buffer overflow.

- Store the legitimate return address (that is, the address of the instruction after the call), and on returns, check before actually returning.

- A stack buffer overflow would be adequate to overwrite the return address on the stack, but not the shadow stack's record of the return address.

- The return address will be checked from the shadow stack and if it doesn't match, appropriate action can be taken. When we run the instrumentation code against bin/ls or any other program, ideally we should not get any error or CFI anomaly.

- But we observe that at some places the return addresses from the shadow stack and the original stack don't match giving us an indication that there's a CFI breach.

  .

- The common calling convention assumes that an invoked function will always return to the address pushed onto the stack by the calling function. However, our experiments have shown that there are a few exceptions violating this calling convention. These exceptions can be categorized into three classes:

  (**Class 1**) - A called function does not return, i.e., the control is transferred out of the function before its return instruction has been reached.

  (**Class 2**) - A function is invoked without explicitly using a call instruction.

  (**Class 3**) - A different return address is computed while the function is running.

SAARTHAK MAHAJAN
04496403114

EXCEPTION HANDILING

Class 1: Setjmp/Longjmp.

For the first class consider a chain of various function calls: A calls B, B calls C, and C calls D. According to the calling convention, all functions must return explicitly after completing their task: D returns to C, C to B, and B to A. However, the system calls setjump and longjump allow to bypass multiple stack frames, which means that before the return instruction of D has been reached, execution is redirected back to A, although B, C, and D have not yet returned. Hence, we expect the return of D, but the program issues the return of A. To avoid a false positive, we use a strategy by popping continuously return addresses of the shadow stack until a match is found or until the shadow stack is empty. The latter case would indicate a ROP attack.

This exception is handled by the following instrumentation code –

```
void ShadowStack::on_call(const ADDRINT call_ins, THREADID tid)
{
        CallStack *stack = (CallStack*)PIN_GetThreadData(tls_call_stack, tid);
        stack->push(call_ins);
}

void ShadowStack::on_ret(const ADDRINT ret_addr, THREADID tid)
{
        CallStack *stack = (CallStack*)PIN_GetThreadData(tls_call_stack, tid);
        while (unlikely( !is_return_addr(stack->pop(), ret_addr) ))
                ;
}
```

Class 2: Unix signals and lazy binding.

Generally, signals are used in Unix-based systems to notify a process that a particular event (e.g., segmentation fault, arithmetic exception, illegal instruction, etc.) have occurred. Once a signal has been received, the program invokes a signal handler. If such a signal handler is implemented through the signal function, then execution is redirected to the handler function without a call instruction. Hence, if the signal handler returns, our instrumentation would raise a false positive, because the return address of the handler function has not been pushed onto the shadow stack. However, the relevant return address is on top of the program stack before the signal handler is executed. To avoid a false positive, we use a signal detector (provided by the Pin API) in order to copy the return address from the program stack onto our shadow stack when a signal is received.

SAARTHAK MAHAJAN
04496403114

```
void ShadowStack::on_signal(THREADID tid, CONTEXT_CHANGE_REASON reason,
        const CONTEXT *orig_ctx, CONTEXT *signal_ctx, int32_t info, void*)
{
        if (likely( reason == CONTEXT_CHANGE_REASON_SIGNAL )) {
                CallStack *stack = (CallStack*)PIN_GetThreadData(tls_call_stack, tid);
                auto signal_ctx_sp = (ADDRINT*)(PIN_GetContextReg(signal_ctx, REG_STACK_PTR));
                stack->push(*signal_ctx_sp);
        }
}

void ShadowStack::on_call_phase2(THREADID tid, _Unwind_Context *uw)
{
        CallStack *stack = (CallStack*)PIN_GetThreadData(tls_call_stack, tid);
        stack->handler_ctx = uw;
}

void ShadowStack::on_ret_phase2(THREADID tid)
{
        CallStack *stack = (CallStack*)PIN_GetThreadData(tls_call_stack, tid);
        PIN_LockClient();

        CallFrame top_frame_copy = stack->pop(); // IPOINT_AFTER is right at the ret instruction, so save the top frame
        ADDRINT catch_addr = ADDRINT(_Unwind_GetIP(stack->handler_ctx)); // IP in catch, i.e. return address
        stack->push(catch_addr);
        stack->push(top_frame_copy);

        PIN_UnlockClient();
}
```

Lazy binding is enabled by default on UNIX-based systems. It decreases the load-time of an application by delaying the resolving of function start addresses until they are invoked for the first time. Otherwise, the dynamic linker has to resolve all functions at load-time, although they may be never called. On our tested Ubuntu system, lazy binding is enforced by a combination of the functions **dl_rtld_di_serinfo** and **dl_make_stackexecutable**, which are both part of the dynamic linker library **linux-ld.so.** After **dl_rtld_di_serinfo** resolves the function's address, it transfers control to the code of **dl_make_stackexecutable** by a jump instruction. Note that **dl_make_stackexecutable** is not explicitly called. However, it redirects execution to the resolved function through a return instruction (rather than through a jump/call). To avoid a false positive, we push the resolved function address onto our shadow stack before the return of **dl_make_stackexecutable** occurs. The resolved address is stored into the %eax register after **dl_rtld_di_serinfo** returns. Hence, we push the %eax register onto our shadow stack when **dl_rtld_di_serinfo** returns legally.

Class 3: C++ Exceptions.

Another type of exceptions are those where the return address is computed while the function executes, whereas the computed return address completely differs from the return address pushed by the call instruction. A typical example for this is C++ exception with stack unwinding. Basically, C++ exceptions are used in C++ applications to catch runtime errors (e.g., division by zero) and other exceptions (e.g., file not found). A false positive would arise if the exception occurs in a function that cannot handle the exception. In such case, the affected function forwards the exception to its calling function. This procedure is repeated until a function is found which is able to handle the exception. Otherwise the default exception handler is

SAARTHAK MAHAJAN
04496403114

called. The invoked exception handler is responsible for calling appropriate destructors for all created objects. This process is referred to as stack unwinding and is mainly performed through the GNU unwind functions **Unwind_Resume** and **Unwind_RaiseException**. These functions make a call to **Unwind_RaiseException_Phase2** that computes the return address and loads it at memory position -0xc8(%ebp), i.e., the %ebp register minus 200 (0xc8) Bytes points to the return address. In order to push the computed return address onto our shadow stack, we copy the return address at -0xc8(%ebp) after Unwind_RaiseException_Phase2 returns legally.

```
void find_cxx_phase2(IMG img, void*)
{
        /*
        when _Unwind_RaiseException_Phase2 returns, its first argument, an _Unwind_Context *,
        contains the IP of the exception handler
        */
        auto rtn = RTN_FindByName(img, "_Unwind_RaiseException_Phase2");
        if (RTN_Valid(rtn))
        {
                RTN_Open(rtn);

                RTN_InsertCall(rtn, IPOINT_BEFORE,
                        AFUNPTR(&on_call_phase2),
                        IARG_THREAD_ID,
                        IARG_FUNCARG_ENTRYPOINT_VALUE, 1, // _Unwind_Context*
                        IARG_END);

                RTN_InsertCall(rtn, IPOINT_AFTER,
                        AFUNPTR(&on_ret_phase2),
                        IARG_THREAD_ID,
                        IARG_END);

                RTN_Close(rtn);
        }
}
```

Now for instrumentation of different programs, we have two instrumentation codes –

1. **allc.so** – It uses a conventional stack, single-threaded instrumentation code that **cannot handle the exceptions** we discussed above and so we may **get false positives** during instrumentation of programs, giving us an indication of a CFI anomaly even though there is none.

2. **shadow_stack.so –** It uses a conventional stack, can handle mutli-threaded programs and also **can handle the exceptions** we discussed above and hence we **won't get any false positives** and will correctly indicate the existence of a CFI anomaly only when there is actually one.

We instrumented various samples/programs for e.g. **longjmp.c, longjmp2.c**, **signal.c**, **c++2.c, threads.c** to demonstrate that **allc.so** cannot handle the exceptions and multi- threaded programs whereas **shadow_stack.so** can handle all exceptions and can be used to detect a Control Flow Integrity(CFI) anomaly or breach in any type of program.

Observations

Samples /Programs	**allc.so** **(cannot handle exceptions)**	**shadow_stack.so** **(can handle exceptions)**
longjmp	**CFI ANOMALY FOUND (**FALSE POSITIVE)	**NO CFI ANOMALY**
longjmp2	**CFI ANOMALY FOUND (**FALSE POSITIVE)	**NO CFI ANOMALY**
signal	**CFI ANOMALY FOUND (**FALSE POSITIVE)	**NO CFI ANOMALY**
C++2	**CFI ANOMALY FOUND (**FALSE POSITIVE)	**NO CFI ANOMALY**
threads	**CFI ANOMALY FOUND (**FALSE POSITIVE)	**NO CFI ANOMALY**

Result

From the above observations, we can infer that **allc.so** cannot handle exceptions and may give **wrong indication of a CFI anomaly** whereas shadow_stack.so will indicate CFI anomaly in any program when the control flow of a program is being diverted elsewhere abnormally and **protect backward edge control flow instructions**.

7.5 Overhead Calculations

We know that the execution of a program with instrumentation takes more time than it's execution without instrumentation. This overhead of instrumentation depends on many factors. The analysis code (the function pointer passed to *InsertCall) contains calls to other functions, loops or conditional statements a context switch might be required. This is something which will increase the execution time of the tool by a lot.

The analysis code we're adding will at the very least cause an increase in execution time equal to the number of instructions in the compiled analysis code, assuming that we're instrumenting on the granularity of an instruction. This will of increase even more if there are loops in your analysis code. *(This is a little simplified, since we assume that every instruction take the same time to execute.)*

Overhead Illustration with Zip files of Linux

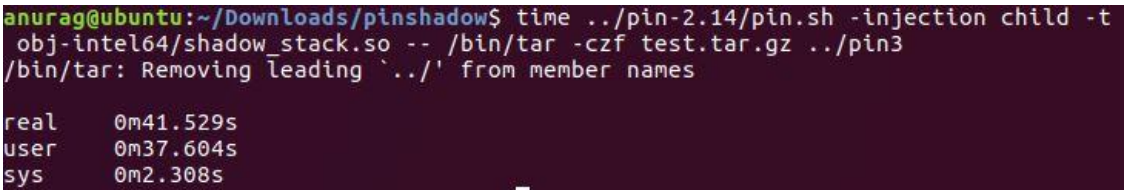
COMPRESSION – WITHOUT INSTRUMENTATION

Total Execution Time – 73.705 seconds

COMPRESSION – WITH INSTRUMENTATION – ALLC.SO

Total Execution Time – 81.330 seconds

COMPRESSION – INSTRUMENTATION - SHADOWSTACK.SO



Total Execution Time – 81.441 seconds

EXTRACTION – WITHOUT INSTRUMENTATION

```
anurag@ubuntu:~/Downloads/pinshadow$ time /bin/tar -xzf test.tar.gz

real    0m9.247s
user    0m5.264s
sys     0m2.376s
```

Total Execution Time – 16.887 seconds

EXTRACTION – WITH INSTRUMENTATION – ALLC.SO

```
anurag@ubuntu:~/Downloads/pinshadow$ time ../pin-2.14/pin.sh -injection child -t
 obj-intel64/allc.so -- /bin/tar -xzf test.tar.gz

real    0m14.700s
user    0m9.604s
sys     0m2.276s
```

Total Execution Time – 26.580 seconds

EXTRACTION – INSTRUMENTATION – SHADOWSTACK.SO

```
anurag@ubuntu:~/Downloads/pinshadow$ time ../pin-2.14/pin.sh -injection child -t
 obj-intel64/shadow_stack.so -- /bin/tar -xzf test.tar.gz

real    0m14.658s
user    0m9.536s
sys     0m2.860s
```

Total Execution Time – 27.054 seconds

Overhead Illustration with Normal Programs

We instrumented programs and computed the overhead factor between normal execution of a program and execution with instrumentation.

1. Prime Number Check

EXECUTION – WITHOUT INSTRUMENTATION

```
anurag@ubuntu:~/Downloads/pinshadow/samples$ time ./Prime1
Enter the number to check prime:8899665522
Number is not prime

real    0m5.891s
user    0m0.000s
sys     0m0.004s
```

Total Execution Time = 5.895 seconds

EXECUTION – WITH INSTRUMENTATION

```
anurag@ubuntu:~/Downloads/pinshadow$ time ../pin-2.14/pin.sh -injection child -t
 obj-intel64/shadow_stack.so  -- ./samples/Prime1
Enter the number to check prime:8899665522
Number is not prime

real    0m51.289s
user    0m0.884s
sys     0m0.292s
```

Total execution time = 52.465 seconds

2. Greatest Common Divisor Calculation

EXECUTION – WITHOUT INSTRUMENTATION

```
anurag@ubuntu:~/Downloads/pinshadow/samples$ time ./gcd
Enter two integers: 888
444
G.C.D of 888 and 444 is 444
real    0m3.427s
user    0m0.000s
sys     0m0.000s
```

Total execution time = 3.427 seconds

EXECUTION – WITH INSTRUMENTATION

```
anurag@ubuntu:~/Downloads/pinshadow$ time ../pin-2.14/pin.sh -injection child -t
 obj-intel64/shadow_stack.so  -- ./samples/gcd
Enter two integers: 888
444
G.C.D of 888 and 444 is 444
real    0m7.551s
user    0m0.844s
sys     0m0.344s
```

Total execution time = 8.739 seconds

SAARTHAK MAHAJAN
04496403114

Observations

We observed various execution times for various scenarios as follows:-

T.E.T = TOTAL EXECUTION TIME| CZF = COMPRESS | XZF = EXTRACT

	WITHOUT INSTRUMENTATION	WITH INSTRUMENTATION (ALLC.SO)	WITH INSTRUMENTATION (SHADOWSTACK.SO)
T.E.T for CZF	73.705 seconds	81.330 seconds	81.441 seconds
T.E.T for XZF	16.887 seconds	26.580 seconds	27.054 seconds
T.E.T for Prime	5.895 seconds	-	52.465 seconds
T.E.T for GCD	3.427 seconds	-	8.739 seconds

- We observe that the **execution times** are in the following order in both, **Compression & Decompression –**

- **WITHOUT INSTRUMENTATION < ALLC.SO < SHADOW_STACK.SO**

- We also observe that the execution of a program with instrumentation takes more time than it's execution without instrumentation.

- The **overhead factor** for a program -

  T.E.T with instrumentation / T.E.T without instrumentation

  For Prime Number Program, Overhead factor = 52.465s/5.895s = 8.89
  For GCD calculation, Overhead Factor = 8.739s/3.427 = 2.55

SAARTHAK MAHAJAN
04496403114

Results

From the above observations, we conclude the following things:-

- The execution of a program with instrumentation takes more time than it's execution without instrumentation.

- The instrumentation done with ***shadow_stack.so*** takes more time than instrumentation done with ***allc.so*** since it has the additional capability of handling all the exceptions discussed in **section 7.4.**

- The overhead of pin tool of programs varies and **depends on the type of program we're executing** –

  Overhead for prime number program – 8.89
  Overhead for GCD calculation – 2.55

SAARTHAK MAHAJAN
04496403114

CHAPTER 8

CONCLUSION

- We studied various memory protection methods – **ASLR, DEP** but, due to their **limitations**, they can be **bypassed** by **Return Oriented programming (ROP)**, in which the attacker takes control of the call stack.

- We also looked at various techniques to defend against ROP but came to a conclusion that we need a **comprehensive security concept** to defend against it.

- **Control Flow Integrity** can be enforced to defend against ROP thereby, protecting the control flow of an application or program. It acts as a strong defence mechanism to restrict the freedom of an attacker.

- Run-time enforcement of CFI has to be done for two types of indirect control flow transfers namely, **forward edge & backward edge transfers.**

- To **enforce CFI** on **backward edge** indirect control flow transfers, we utilized **dynamic binary instrumentation** provided by **Intel Pin Tool**.

- For the **backward edg**e, a context-sensitive approach that enforces **stack integrity** and at the same time using appropriate exception handling techniques guarantees full protection.

- We implemented a mechanism called **shadow stack** capable of handling all classes of exceptions and ensuring the protection of **backward edge transfers.**

- We also observed that the **overhead** of instrumenting programs with instrumentation code, having the additional capability of handling exceptions is **maximum.**

- The protection of **forward edge transfers** can be done by using an equivalence check between the valid set of targets and observed targets.

- The implementation for forward edge transfer protection hasn't been realized yet, but once it is, it will provide us the **complete solution** against various **Control Flow Attacks** especially, Return-Oriented-Programming.

SAARTHAK MAHAJAN
04496403114

REFERENCES

- **Control-Flow Integrity Principles, Implementations, and Applications, 2005 (**MARTIN ABADI, University of California, Santa Cruz and Microsoft Research, Silicon Valley MIHAI BUDIU, Microsoft Research, Silicon Valley U´ LFAR ERLINGSSON Microsoft Research, Silicon Valley and JAY LIGATTI University of South Florida)
  https://users.soe.ucsc.edu/~abadi/Papers/cfi-tissec-revised.pdf

- **Practical Control Flow Integrity & Randomization for Binary Executables, 2013** (CHAO ZHANG, Peking University ,China TAO WEI, University Of California Lei Duan, Stony Brook University WEI ZOU, Peking University)
  www.ieee-security.org/TC/SP2013/papers/4977a559.pdf

- **ROPdefender, 2015**
  (Lucas Davi, Technische Universität Darmstadt, Germany, Ahmed-Reza-Sadeghi, Ruhr-Universität Bochum, Germany)
  https://pdfs.semanticscholar.org/1156/5da78c06a94c1fc8a0ff3a8d710cb9a5d450.pdf

- **Pin Tool 2.14 User Guide**
  https://software.intel.com/sites/landingpage/pintool/docs/76991/Pin/html

- **Return-Oriented-Programming**
  https://en.wikipedia.org/wiki/Return-oriented_programming

- **Control-Flow-Integrity**
  https://www.trust.informatik.tu-darmstadt.de/research/projects/current-projects/control-flow-integrity/