

Computer and Network Security via Program Analysis

Monirul Sharif

Guest Lecture

CS-6265

College of Computing

Georgia Institute of Technology

Date: 09/20/2006

Introduction

- We will cover several topics related to **program analysis** for security.
- Today, program analysis techniques for defending against attacks.
- Next class will cover how to use program analysis on malware.

Vulnerabilities and Exploits

- Vulnerability
 - A weakness in a system caused by a **flaw** that allows an attacker to violate *integrity, confidentiality, authenticity* or *availability* of a system.
 - The *flaws* are usually *programmer errors* or **bugs**.
- Exploits
 - A software, a portion of data, or a sequence of commands that is made to take advantage of a **vulnerability** in order to violate *integrity, confidentiality, authenticity* or *availability* of a system.
 - Used as part of an attack on a system.

Malware

- Software designed to infiltrate or damage computer systems – **“Malicious Software”**
 - Computer viruses, worms, trojan horses, spyware, adware
 - May use exploits as a means infiltrating a computer automatically.
- Can be infectious
 - Viruses, worms spread from file to file and computer to computer respectively.
- Disguised or using concealment
 - Trojan horses are disguised malware.
- Malware for profit by hackers
 - Spyware, adware, loggers and diallers.
 - Botnets – controlled army of compromised hosts
- We will be looking at malware analysis in details in the next class.

Software Vulnerabilities

- Bugs are common in software
 - Implementation error by a programmer
 - Bugs are hard to find in large software
 - Huge budget on testing, almost 80% of software development. But still bugs exist.
- Some bugs are considered security flaws
- Bugs -> Security Flaw -> Vulnerability -> Exploit -> Attack
- Take a look at Microsoft Software
 - Don't blame Microsoft.
 - Microsoft continues to spend a lot on testing
 - The fact that Microsoft software is used by millions, reveal more bugs, and attract more attackers.

An Example Vulnerable Program

```
bool checkpassword()
{
    bool passwordok = false;
    int userid = getcurrentuserid();
    char password[10];

    puts("Enter password:");
    gets(password);

    if (strcmp(getpassword(userid), password)==0)
        passwordok = true;

    return passwordok;
}

int main(int argc, char *argv[])
{
    // first get user id
    ... ..
    // then check password
    if (!checkpassword) exit(1);
    ...
}
```

- Where is the bug in the program?
- How can it be exploited?

Buffer Overflows

```
bool checkpassword()
{
    bool passwordok = false;
    int userid = getcurrentuserid();
    char password[10];

    puts("Enter password:");
    gets(password);

    if (strcmp(getpassword(userid),
              password)==0)
        passwordok = true;

    return passwordok;
}
```

Bug: Unbounded input in a bounded buffer

- This stack overflow vulnerability allows the following attacks:
 - Gain privileged access with unprivileged user id and password
 - Most frightening – arbitrary code execution
- Let's see how it works briefly.

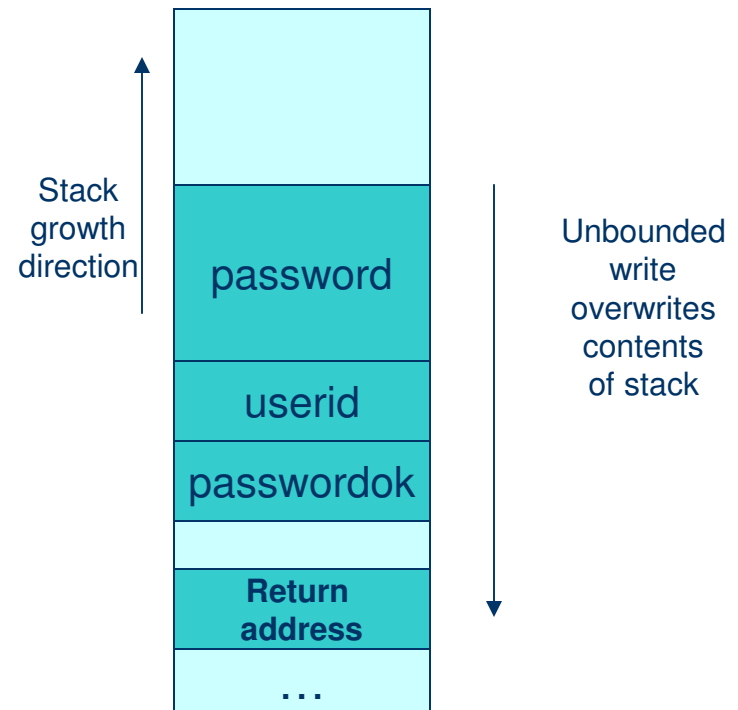
Buffer Overflows

```
bool checkpassword()
{
    bool passwordok = false;
    int userid = getcurrentuserid();
    char password[10];

    puts("Enter password:");
    gets(password);

    if (strcmp(getpassword(userid),
              password)==0)
        passwordok = true;

    return passwordok;
}
```



Structure of
Data on stack

Buffer Overflows

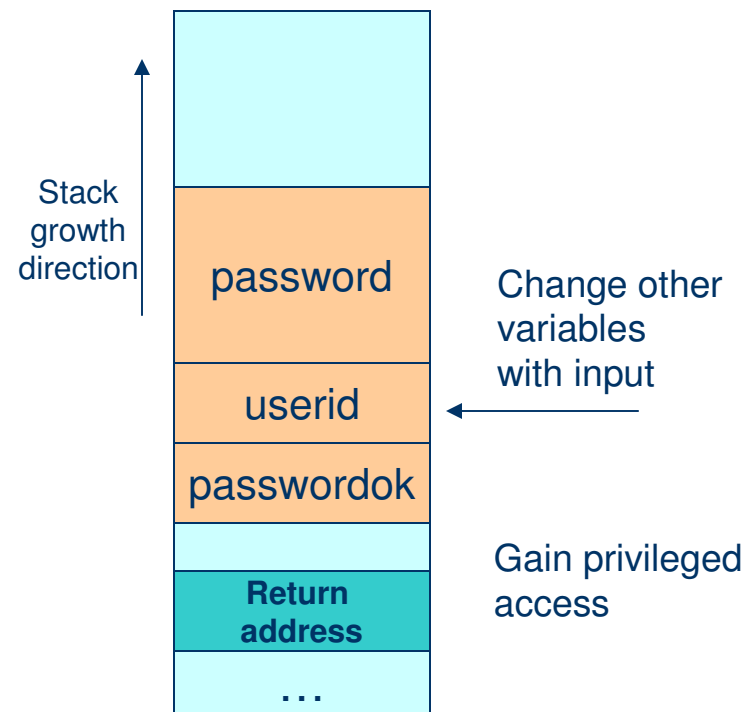
```
bool checkpassword()
{
    bool passwordok = false;
    int userid = getcurrentuserid();
    char password[10];

    puts("Enter password:");
    gets(password);

    if (strcmp(getpassword(userid),
              password)==0)
        passwordok = true;

    return passwordok;
}
```

- The exploit is a specially crafted input larger than 10 bytes.
- Exploit can change local variables to let an attacker gain privileged access.



Structure of Data on stack

Buffer Overflows

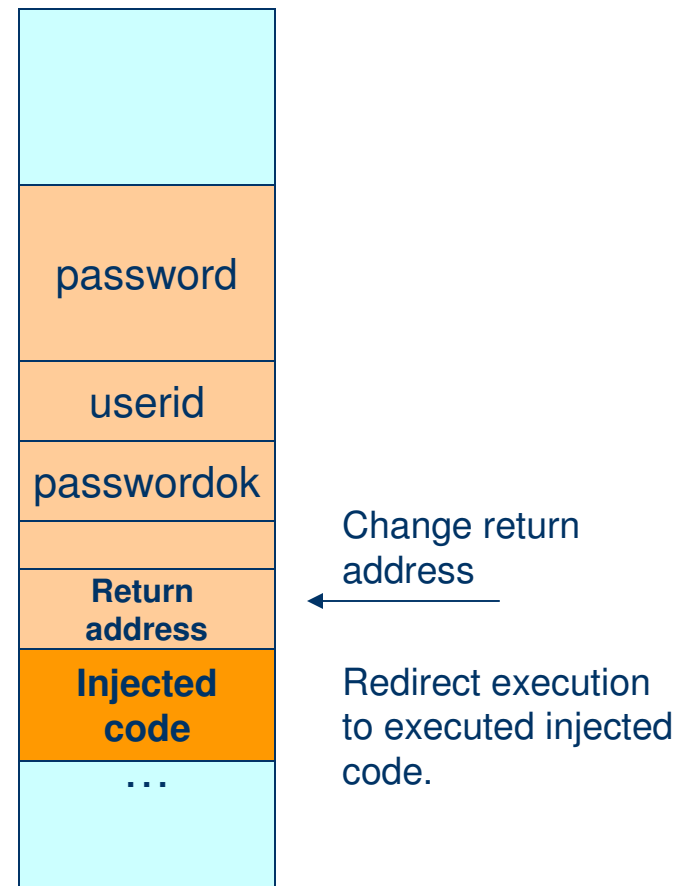
```
bool checkpassword()
{
    bool passwordok = false;
    int userid = getcurrentuserid();
    char password[10];

    puts("Enter password:");
    gets(password);

    if (strcmp(getpassword(userid),
               password)==0)
        passwordok = true;

    return passwordok;
}
```

- The exploit can contain injected code and change return address to point to it.
- Injected code can contain a **shellcode**, to provide root access to the attacker.



Vulnerabilities in the Wild

- Most common exposed vulnerabilities include
 - Buffer overflows, Heap overflows, Format Strings, Integer overflows etc.
- Disclosure of common vulnerabilities
 - Security alerts are posted regularly by experts
 - <http://cve.mitre.org/> and <http://www.us-cert.gov/>
 - Software developers become aware and **fix** the vulnerabilities by making downloadable **patches** available
 - Question: A hacker may come to know of a published vulnerability and exploit it. Why expose publicly?
- Severity of vulnerability depends on
 - How much an attacker can achieve by exploiting it
 - How widespread the vulnerable application is.

Threats and Attacks

- Common attacks are:
 - DoS – very simple, just corrupt memory to crash application
 - Privilege escalation – increase access privilege maliciously
 - Privacy attack – divulge private information
 - Arbitrary code execution – execute foreign code.
- Most Risky Vulnerabilities allow arbitrary code execution
 - Hackers can gain access to remote systems and take complete control.
 - Can be used to automatically spread malware. Fast automated spreading.
 - Compromised computers become zombies, bots etc. that are controlled by an attacker to serve other purposes
- Worms and Bots
 - May contain code that exploits remote vulnerabilities to spread themselves.
 - Once exploitation is successful, infect host with malware.

Where does program analysis come in?

- Program Analysis
 - *Software Engineering* approach that has become increasingly important in security. A lot techniques derived from *Compilers*.
 - Used in SE for
 - Testing and debugging
 - Formal methods – model checking, verification of properties of software (undecidable, more later)
 - Profiling and a lot more
 - Used extensively in Compilers for code optimization.
- Its place in security
 - Why do we need it? Vulnerabilities exist in programs. *Mal-ware* are programs written for malicious purposes.
 - Some uses -
 - Vulnerability identification (a lot like bug finding) and hardening Software (protect from exploitation)
 - Intrusion detection & intrusion prevention
 - Malware detection and Malware analysis (reverse engineering)

Program Analysis Approaches

- Program analysis is almost always done using automated tools. Human beings direct analysis.
- Used on:
 - Source code: Available source is analyzed.
 - More meaningful analysis can be done.
 - Binary: Executable version is analyzed
 - Used in cases where source code is not available.
 - Has become very important in security.
 - Very hard, a lot of information is lost.
- Two approaches
 - Static Analysis
 - Analysis is done without executing program
 - Dynamic Analysis
 - Executing program is observed for analysis

Static Analysis

- Analysis is done without executing program
 - All possible execution patterns are taken into account
 - Actual run-time behavior is covered but over-assumption remains (imprecise)
 - Sound and formal methods can be applied.
- What sorts of analysis?
 - Several well-defined techniques exist for
 - *control-flow analysis, data-flow analysis, control-dependence analysis, data-dependence analysis.*
 - Usually program code is represented as graphs. Algorithms are applied on graph data structures for automated analysis.
- Limitations
 - Exact execution pattern is hard to know.
 - Most properties that are analyzed.
 - Obfuscation may hinder static analysis. Used extensively by malware (next class)

Dynamic Analysis

- Executed program is analyzed
 - One execution gives a partial view, therefore several attempts are required
 - Execution gives the actual behavior of the program (precise)
- How performed:
 - Observable behavior can be analyzed
 - System calls
 - I/O, file access etc.
 - Snippets of code is inserted so that more run-time information can be gathered (Instrumentation)
 - Code can write logs, check for conditions, or change execution as required.
 - Tools – Valgrind, DynInst etc.
- Limitations
 - Gathering complete picture of program behavior may become infeasible

Binary Analysis

- Has become very important in security
 - The need to protect/secure legacy and commercial software, where source code is not available.
 - Malware is found in binary form
 - Reverse engineering to understand how malware works.
- Very important for malware detection and analysis
 - Generate signatures for detection
 - Generate behavior patterns.
 - Understanding how malware works
 - *We will cover this huge topic in the next class.*
- Has been used by antivirus companies for a long time. But now “hot” for security research.

Example – Control-flow Analysis

- Here's an example of static analysis on code
- Control-flow analysis is done by first finding the CFG (Control-flow Graph)
 - A “directed graph”.
 - Each node is usually a *basic-block* of code (sequentially executing block of statements)
 - Possible control-flow is shown by directed edges between the nodes.
 - Loops, conditionals are represented
- Identify individual functions in the program
 - For each function find CFG – Intra-procedural CFG
 - Make Inter-procedural CFG - Connect CFG's using interprocedural control-flow (function calls and return sites)
- CFG is the building block of almost all other program analysis techniques.

Can we find vulnerabilities?

- A very hard problem
 - Theoretical results show, determining if a program will show a specific run-time error is *undecidable* (derived from the *halting problem*). Vulnerabilities are the same.
 - Also, checking a property of a language is *undecidable*.
- Even if we know the vulnerability type, there are approximate solutions
 - Approximation gives rise to
 - False positives – A warning of a vulnerability even though its not.
 - False negatives – Missing a vulnerability.
- Several other problems make it hard
 - Pointers. Pointer analysis is imprecise

Can we find vulnerabilities?

- Let us do an exercise. Think about how Buffer overflows can be detected.
 - First let us think in terms of static analysis.
 - Now let us think of dynamic analysis methods

Can we find vulnerabilities?

- Many tools exist. Some identify, some protect without identifying.
 - An example is ProPolice
 - Analyzes code statically
 - Uses heuristics to detect possible bugs
 - A lot of false positives, and false negatives
 - Stackguard
 - A dynamic method
 - Uses special canary values around buffers. Detects at run-time when overwrite occurs.
 - *It does not find a vulnerability*, but will protect a program if one exists
- In the Practical World
 - Human experience and expertise and hours of hard work.
 - Many security experts use automated tools that aid in search by finding it potential points.

Program Analysis for Defense

- New types of vulnerabilities may be found, so we may need security without identifying them.
- Now let us consider another direction of securing applications
 - Suppose that we do not know about vulnerabilities
 - Suppose that we only know about different types of attacks.
 - What can we do?
- Intrusion Detection
 - Will be covered in details throughout the course.
 - We will only consider instances where program analysis is used.

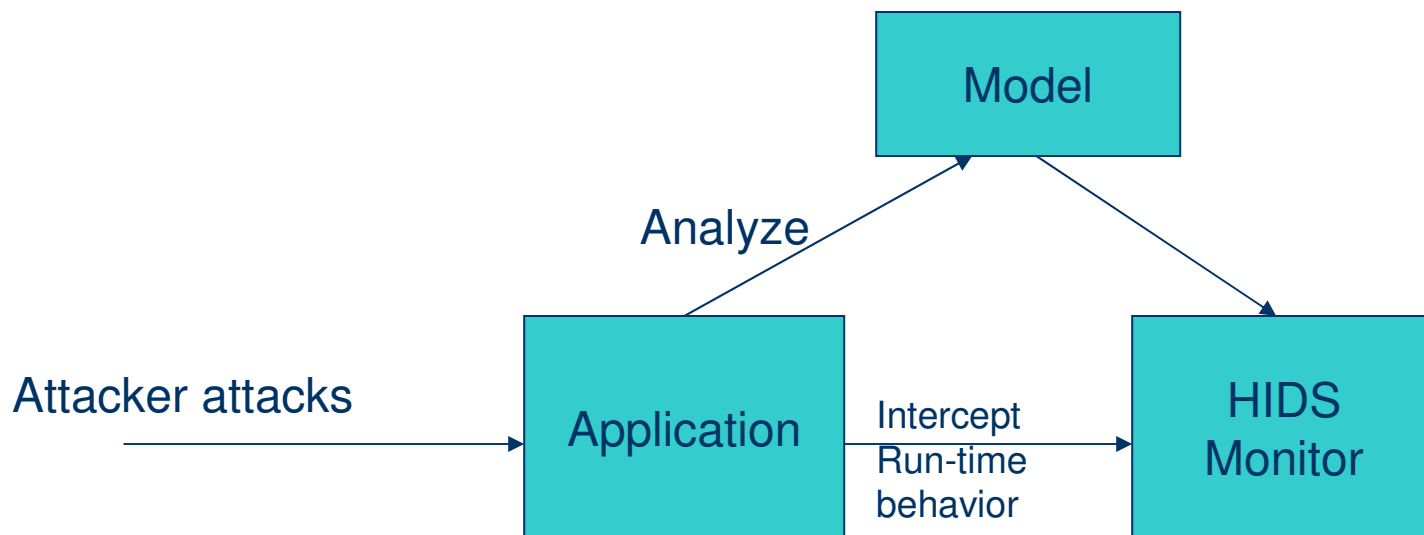
Host based Intrusion Detection

- Monitor activity and raise alert for a possible intrusion (violation in security)
- Detection is based on activity at a host
 - Different from Network-based Intrusion Detection that observes network traffic only
 - Activity may include
 - Application activity
 - Network communication from the host
 - System logs etc.
- Again two classifications
 - Misuse detection – use signatures of known attacks
 - Anomaly detection – use model of “normal” behavior
- We will consider anomaly HIDS that use **program analysis**

Host based Intrusion Detection (cont)

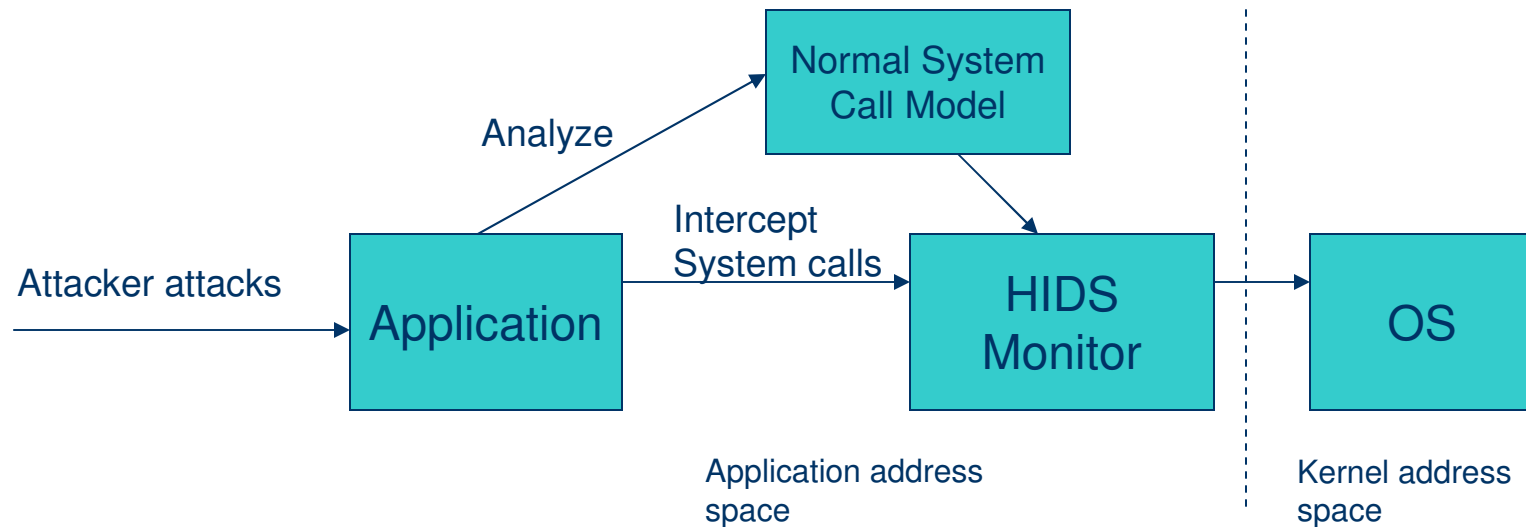
- Misuse detection – use signatures of known attacks
 - Signature can be constructed from exploit code.
 - Can be constructed from semantics of attack (steps in the exploit)
 - Our next class will look at malware analysis.
- Anomaly detection – use model of “normal” behavior
 - How do we define normal behavior?
 - We use *program analysis* to find the possible execution behavior of the application (normal behavior) and represent it using a “model”
 - An anomaly, which is a deviation, raises an alarm. Possible attack

Host-based Anomaly IDS



- A lot of products are available.
- IPS (Intrusion Prevention Systems – IDS with automatic preventive steps, rather than just raising alarms)
- Has been an area of intense research throughout the years, and still has work to be done.
- What do you base your model on? Most prominent in research - **System call sequences.**

System-call sequence based HIDS



- System calls are easily observable, OS can help intercepting them
- Useful attacks require system calls to be invoked – opening files, execve etc.
- Efficient to intercept, no extra overhead
- Main research aspects are how to create a “model”. Extensive research work in the literature.

Building Models of System-call Sequences

- Model representation
 - A sliding window of system call sequences (finite length possible sequences)
 - Automaton based approaches – take program control-flow into account
 - build NFA (Non-deterministic finite Automata)
 - PDA (push-down automata) or other complex models
- Static Analysis
 - Analyze program, and find system-call points in code
 - Model is constructed from “all-possible” valid execution paths.
 - Due to safe coverage, “zero false-positives”.
 - Due to imprecision attacks may be missed, false negatives.
- Dynamic Analysis
 - Build model from system call trace of executions.
 - Learn from several runs.
 - Due to partial coverage, false positives are appear
 - Due to better precision, might catch more attacks

Example of Intrusion Detection

```
bool checkpassword()
{
    bool passwordok = false;
    char password[10];

    puts("Enter password:");
    gets(password); //read

    if (strcmp(getpassword(), password)==0) {
        passwordok = true;
        writestatus(); //write
    }
    return passwordok;
}

int main(int argc, char *argv[])
{
    // open files    // open
    ...
    // then check password
    if (!checkpassword) exit(1); //exit
    // close files  // close
}
```

Allowed sequences:

Open, read, exit

Open, read, write, exit

Open, read, close ...

Open, read, write, close ...

Example of Intrusion Detection

```
bool checkpassword()
{
    bool passwordok = false;
    char password[10];

    puts("Enter password:");
    gets(password); //read

    if (strcmp(getpassword(), password)==0) {
        passwordok = true;
        writestatus(); //write
    }
    return passwordok;
}
```

```
int main(int argc, char *argv[])
{
    // open files    // open
    ...
    // then check password
    if (!checkpassword) exit(1); //exit
    // close files  // close
```

Allowed sequences:

Open, read, exit

Open, read, write, exit

Open, read, close ...

Open, read, write, close ...

Monitored sequences:

Open, read, **execve**
(intrusion detected)

Attack code
(execve, etc)

Mimicry Attacks

- Evading Detection
 - Attacker cloaks execution of attack by revealing behavior that IDS cannot find an anomaly.
- Mimicry attacks on system call sequences
 - An attacker knows the model used by IDS
 - Inserts extraneous system calls to form sequences that follow the model
 - Eventually the IDS cannot detect, and attacker succeeds in achieving goal
 - Actually it is hard because required system calls must exist

Example of Mimicry Attacks

```
bool checkpassword()
{
    bool passwordok = false;
    char password[10];

    puts("Enter password:");
    gets(password); //read

    if (strcmp(getpassword(), password)==0) {
        passwordok = true;
        writestatus(); //write
    }
    return passwordok;
}
```

```
int main(int argc, char *argv[])
{
    // open files    // open
    ...
    // then check password
    if (!checkpassword) exit(1); //exit
    // close files  // close, execve
}
```

Allowed sequences:

...

- Open, read, close, execve
- Open, read, write, exit
- Open, read, close ...
- Open, read, write, close ...

Monitored sequences:

Open, read, close, execve
(intrusion not detected)

Attack code
(execve, etc)

Project Ideas

- Develop IDS
 - Work with vulnerable programs and exploits in VMWare
 - Use “ptrace” to intercept system calls
 - Create model and monitor program to detect attacks
- Vulnerability detection and removal.
- Malware related projects will be pointed out in the next class.