WILLIAM & MARY
CHARTERED 1693

CSCI 445: Mobile Application Security

Lecture 21

Prof. Adwait Nadkarni

Derived from slides by William Enck

Announcements

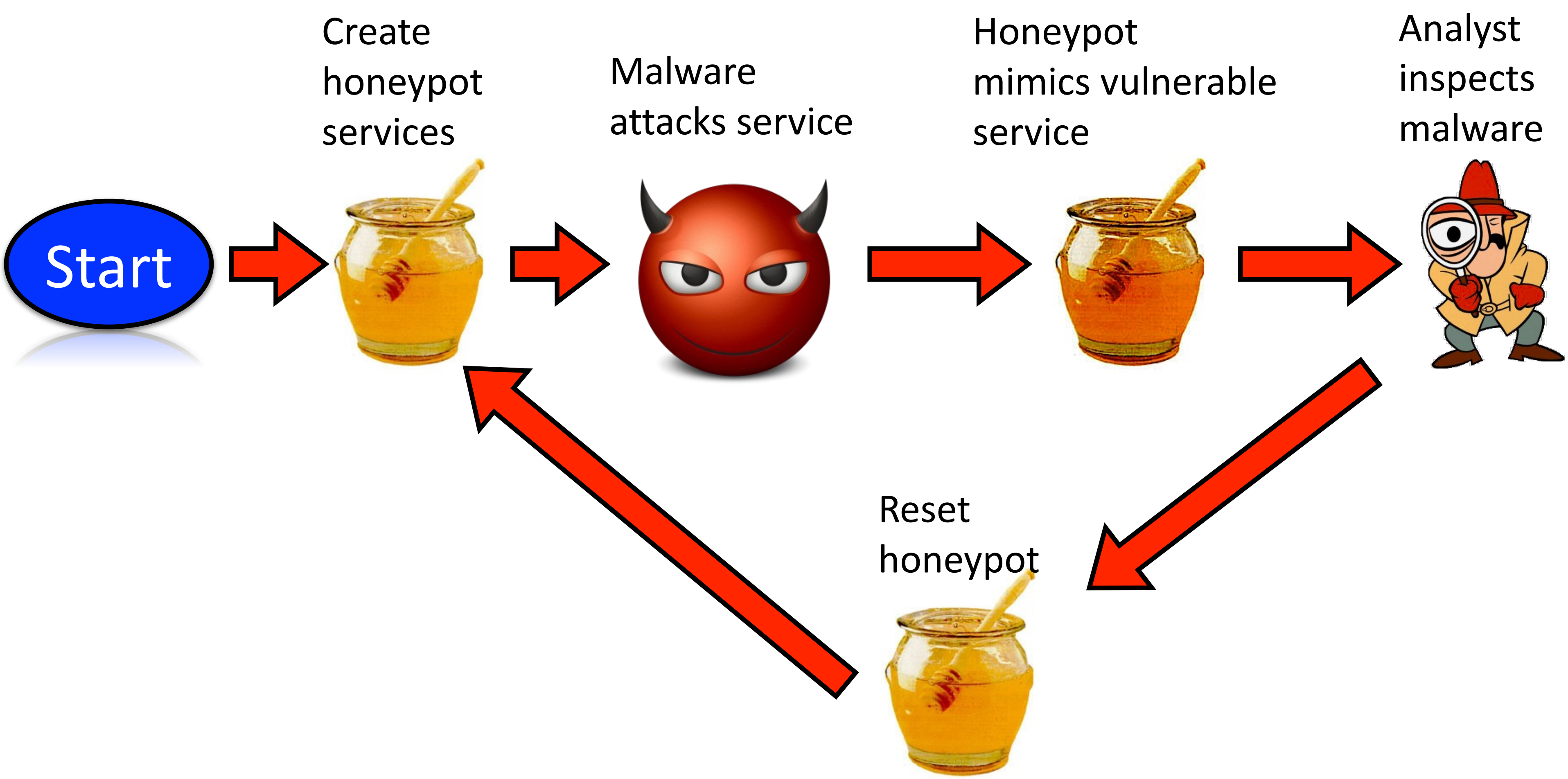
- Will release HW5 today
- Project presentations and final review split across 2 lectures?
  - Lots of interest, need to give each team sufficient time to present + for questions
  - Spreads the questions regarding the finals out too
- ***Ask for an extension for Milestone 4 if you need one***

How do we learn about and study malware?

Honeypots: *what*

- **Honeypot:** a controlled environment constructed to trick malware into thinking it is running in an unprotected system
  - collection of decoy services (fake mail, web, ftp, etc.)
  - decoys often mimic behavior of unpatched and vulnerable services

Example Honeypot Workflow



Honeypots: *why*

- Three main uses:
  - forensic analysis: better understand how malware works; collect evidence for future legal proceedings
  - risk mitigation:
    - provide "low-hanging fruit" to distract attacker while safeguarding the actually important services
    - tarpits: provide very slow service to slow down the attacker
  - malware detection: examine behavior of incoming request in order to classify it as benign or malicious

Honeypots: *types*

- Two main types:
  - **Low-interaction:** emulated services (e.g., a script)
    - inexpensive
    - may be easier to detect
  - **High-interaction:** no emulation; honeypot maintained inside of real OS
    - expensive
    - good realism
      - But not too real → bad form to actually help propagate the worm (*legal risks!*)

honeyd

- Open-source virtual honeynet
  - creates virtual hosts on network
  - services actually run on a single host
  - scriptable services

honeyd example:

FTP service (ftp.sh)

```
echo "$DATE: FTP started from $1 Port $2" >> $log
echo -e "220 $host.$domain FTP server (Version wu-2.6.0(5) $DATE) ready."
...
case $incmd_nocase in

        QUIT* )
           echo -e "221 Goodbye.₩r"
           exit 0;;
        SYST* )
           echo -e "215 UNIX Type: L8₩r"
           ;;
        HELP* )
           echo -e "214-The following commands are recognized (* =>'s unimplemented).₩r"
           echo -e "   USER    PORT    STOR    MSAM*   RNTO    NLST    MKD     CDUP₩r"
           echo -e "   PASS    PASV    APPE    MRSQ*   ABOR    SITE    XMKD    XCUP₩r"
           echo -e "   ACCT*   TYPE    MLFL*   MRCP*   DELE    SYST    RMD     STOU₩r"
           echo -e "   SMNT*   STRU    MAIL*   ALLO    CWD     STAT    XRMD    SIZE₩r"
           echo -e "   REIN*   MODE    MSND*   REST    XCWD    HELP    PWD     MDTM₩r"
           echo -e "   QUIT    RETR    MSOM*   RNFR    LIST    NOOP    XPWD₩r"
           echo -e "214 Direct comments to ftp@$domain.₩r"
           ;;
```

Examining Malware

- **Trace system calls:**
  - most OSes support method to trace sequence of system calls
    - e.g., ptrace, strace, etc.
    - Or, monitor API calls (recall: *hooks in ASM, TaintDroid)*
  - all "interesting" behavior (e.g., networking, file I/O, etc.) must go through system calls
  - capturing sequence of system calls (plus their arguments) reveals useful info about malware's behavior
  - Question: *Can Antiviruses do this on smartphones?*

Tracing System Calls

```
% strace ls
open("/proc/filesystems", O_RDONLY)     = 3
fstat(3, {st_mode=S_IFREG|0444, st_size=0, ...}) = 0
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f88345a4000
read(3, "nodev\tsysfs\nnodev\trootfs\nnodev\tb"..., 1024) = 346
read(3, "", 1024)                       = 0
close(3)                                = 0
munmap(0x7f88345a4000, 4096)            = 0
open("/usr/lib/locale/locale-archive", O_RDONLY) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=2772576, ...}) = 0
mmap(NULL, 2772576, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f88330f9000
close(3)                                = 0
ioctl(1, SNDCTL_TMR_TIMEBASE or TCGETS, {B38400 opost isig icanon echo ...}) = 0
ioctl(1, TIOCGWINSZ, {ws_row=24, ws_col=80, ws_xpixel=0, ws_ypixel=0}) = 0
open(".", O_RDONLY|O_NONBLOCK|O_DIRECTORY|O_CLOEXEC) = 3
fcntl(3, F_GETFD)                       = 0x1 (flags FD_CLOEXEC)
getdents(3, /* 36 entries */, 32768)    = 1104
getdents(3, /* 0 entries */, 32768)     = 0
close(3)                                = 0
fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 1), ...}) = 0
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f88345a4000
write(1, "mail  R  shared  tmp  work\n", 27) = 27
close(1)                                = 0
munmap(0x7f88345a4000, 4096)            = 0
close(2)                                = 0
exit_group(0)                           = ?
```

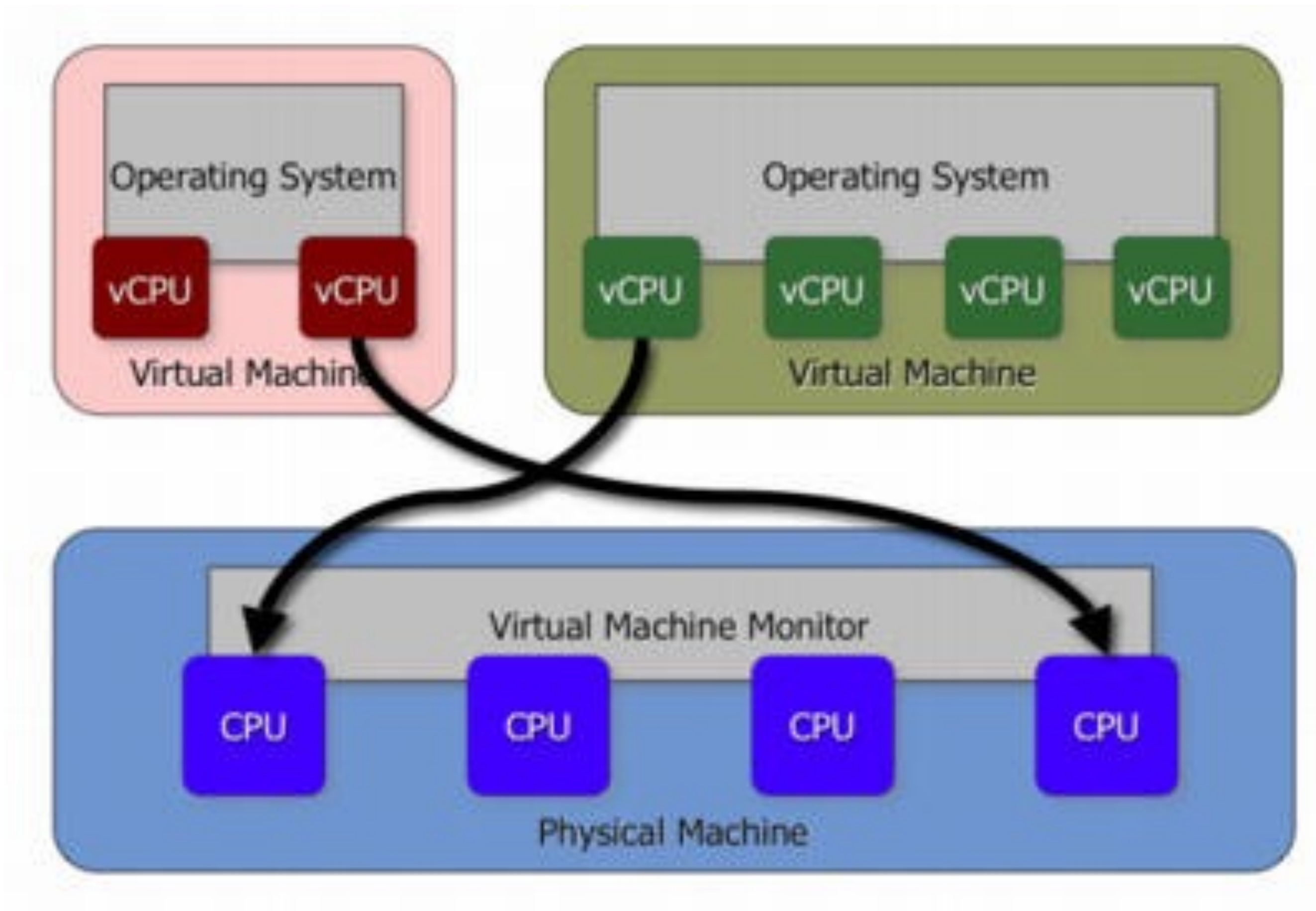
Examining Malware

- **Observe filesystem changes and network IO:**
  - “diff” the filesystem before and after
    - which files are the malware reading/writing?
  - capture network packets
    - to whom is the malware communicating

Challenges

- Honeypot *must resemble actual machine*
  - simulate actual services (Apache, MySQL, etc.)
  - but not too much... bad form to actually help propagate the worm (*legal risks!*)
- Some worms do a reasonably good job of detecting honeypots

Virtual Machines



Virtual Machines

- **Virtual machine:** isolated virtual hardware running within a single operating system
  - i.e., a software implementation of hardware
  - usually provides emulated hardware which runs OS and other applications
  - i.e., a computer inside of a computer
- What's the point?
  - extreme software isolation -- programs can't easily interfere with one another if they run on separate machines
  - much better hardware utilization than with separate machines
  - power savings
  - easy migration -- no downtime for hardware repairs/improvements

Honeypots and Virtual Machines

- Most virtual machines provide checkpointing features
  - **Checkpoint** (also called **snapshot**) consists of all VM state (disk, memory, etc.)
  - In normal VM usage, user periodically creates snapshots before making major changes
  - Rolling back (“restoring”) to snapshot is fairly inexpensive
- **Checkpointing features are very useful for honeypots**
  - Let malware do its damage
  - Pause VM and safely inspect damage from virtual machine monitor
  - To reset state, simply restore back to the checkpoint

Honeypots and Virtual Machines

- Virtual Machines are also very useful for analyzing malware (can *debug* malware):
  - execute malware one instruction at a time
  - pause malware
  - easily detect effects of malware by looking at "diffs" between current state and last snapshot
  - execute malware on one VM and uninfected software on another; compare state

Recall: Evasive Malware

- Lots of research into detecting when you're in a virtual machine (i.e., *to prevent dynamic analysis*)
  - examine hardware drivers
  - time certain operations
  - look at ISA support
- Malware does this too!
  - if not in VM, wreak havoc
  - if in VM, self-destruct
- So, to be malware-free, *why not run your host in a virtualized environment?*

Detecting *Mobile Malware*

Traditional detection systems?

- Can we use antivirus software built for desktops?
  - Android/iOS malware is increasingly *mobile-specific*
- Important to understand the attacker's goals and abilities
  - Stealing private information
  - Costing money (e.g., premium messages)
  - Remote control
  - ...
- In many cases, *no exploits*, simple permission misuse
- *We need techniques that detect traditional malware (e.g., rootkits), and also tailored techniques for smartphones.*

Attack Vectors

- Comprehensive characterization: Zhou and Jiang [1]
- *How does malware get on our devices?*
  - Mostly using *social engineering*
- Malware relies on the user to initiate installation
  1. Repackaging
     a. All at once
     b. Runtime download of payload
  2. Drive-by download

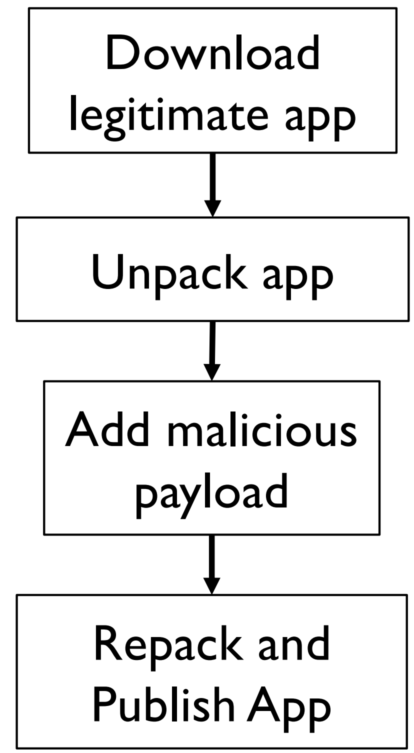
[1] Zhou, Yajin, and Xuxian Jiang. "Dissecting android malware: Characterization and evolution." In *Security and Privacy (SP), 2012 IEEE Symposium on*, pp. 95-109. IEEE, 2012.

Repackaged Malware

- Where is it published?
  - Third-party stores (generally)
  - Official Stores (e.g., Google Play)

1. To find repackaged apps in ***third-party stores***
   i. Look for an app with the same package name as an *official app*
   ii. Static/dynamic analysis: Is the difference benign?

General Approach

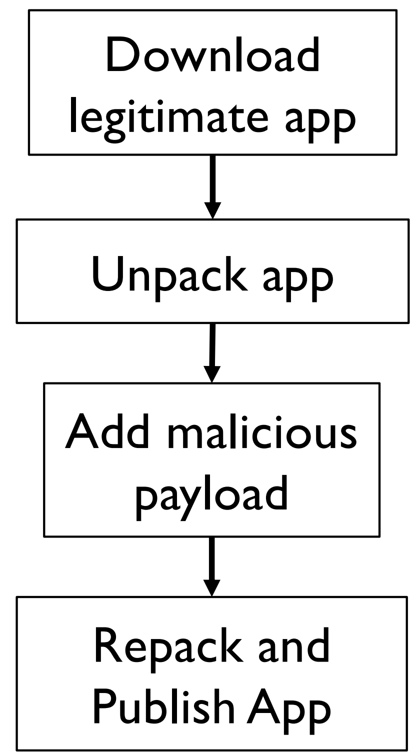


Repackaged Malware

- Where is it published?
  - Third-party stores (generally)
  - Official Stores (e.g., Google Play)

2. To find repackaged apps on ***Google Play***
   i. Can't use package names: package names are *unique!*
   ii. Detect similarity using *metadata*

General Approach



Automated similarity analysis using metadata

- Text analytics (e.g., bag of words)
  - Titles, descriptions, developer names (in that order)
- *Fuzzy* image matching:
  - icons

Original App		*Impostor App*
The Coupons App	**Title**	The Coupons App
Most Popular Download	**Developer Name**	*Most Popular Downloads*
thecouponsapp.coupon	**Package Name**	*thecouponsapp.dailydeals*
	**Icon**	
10 - 50 million	**Downloads**	0.5 - 1 million

- Effective for detecting *grayware*
  - May also help detect retargeted malware (e.g., free repackaged versions of paid apps)

Andow, Benjamin, Adwait Nadkarni, Blake Bassett, William Enck, and Tao Xie. "A study of grayware on google play." In *Security and Privacy Workshops (SPW), 2016 IEEE*, pp. 224-233. IEEE, 2016.

Malware Detection in Practice

- **Target mobile-specific objectives:**
  - Privilege Escalation: Generally, gain *root* privilege
    - Execute one or more root exploits
    - Many exploits are publicly available! (e.g., *towelroot*)
  - Remote Control: Botnets!
  - Charging users: Premium messages, phone calls
  - Stealing private data
  - ...

Malware Detection in Practice

- **Know the limitations of analysis**
  - Malware often hides behavior to evade static analysis
    - Code obfuscation
    - Encrypting code/ root exploits
      - Storing it as an asset
    - Dynamically updating the malicious app
    - JNI
- Problem: *Some of these behaviors are also exhibited by benign apps!*

Malware Detection in Practice

- Boils down to a **classification problem**
- Typical approach:
  1. Select interesting features/feature-types
  2. Train with known malware/benign apps: Use lightweight static analysis to extract features
  3. Use machine learning on feature vectors to classify as benign or malicious
  4. *Test on unlabeled samples*
- VirusTotal: Aggregates results from over 70 virus scanners (most of these are signature based)

Feature Selection

1. **AndroidManifest.xml:**
   - Requested permissions: Sensors, sensitive/private API
   - App components
   - Intent Filters
2. **Disassembled code:** Sensitive API calls
   - APIs for which permissions *have not* been requested. Why?
     - *Sign of potential privilege escalation*
   - Permissions actually *used*
   - Suspicious APIs: get IMEI, dynamic code loading
   - URLs/host names for network communication. Why?
     - *Attributing/ connecting malware samples*

Advantages

- Automated
- Explainable (sometimes; e.g., DREBIN)
- It *generalizes:* Why is this important?
  - Need to detect *variations* of malware
  - More robust against typical evasive maneuvers (e.g., dynamic code loading, obfuscation, etc.)
    - Relies on *a diverse array of features*

Limitations

- Craft **adversarial examples:** Make changes that evade detection, but without changing behavior
  1. Adversary has your model
  2. Adversary does not have model, *but*, can query for *malicious/benign*, and *confidence score*
  3. Adversary can query for *malicious/benign*

Difficulty for the adversary (increasing from 1 to 3)

- How to get labeled data?: *like everyone does*
  - Query VirusTotal! (or specific scanners you want to evade)
- **Is #3 feasible?:** Train a neural network on this labeled dataset.
  - *Key property: Transferability: If an adversarial sample evades my model, it will also evade other similar models*

The End