31st IEEE Symposium on Security & Privacy

TaintScope: A Checksum-Aware Directed Fuzzing Tool for Automatic Software Vulnerability Detection

Tielei Wang[1], Tao Wei[1], Guofei Gu[2], Wei Zou[1]
[1]Peking University, China
[2]Texas A&M University, US

Outline

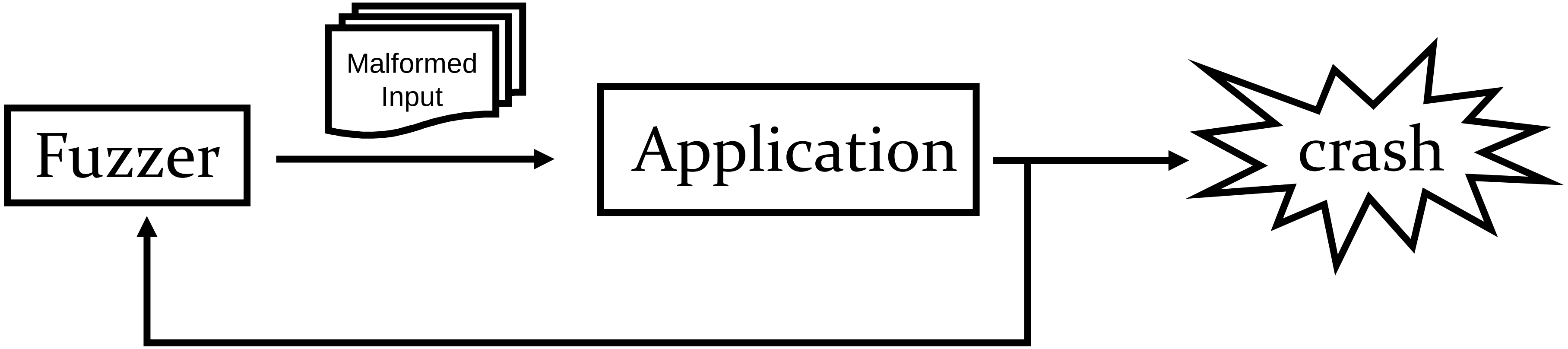
- **Introduction**
  - Background
  - Motivation
- **TaintScope**
  - Intuition
  - System Design
  - Evaluation

Microsoft Google Adobe ......

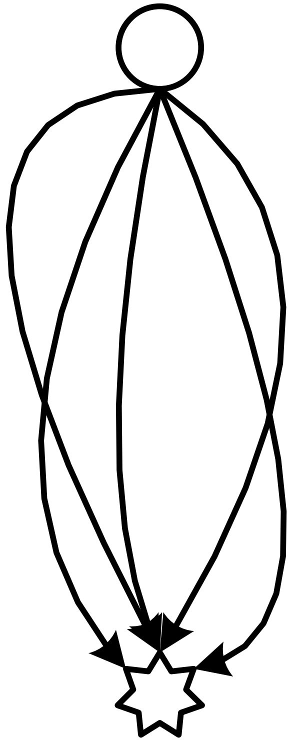
- **Conclusion**

Fuzzing/Fuzz Testing

- Feed target applications with malformed inputs e.g., invalid, unexpected, or random test cases
  - Proven to be remarkably successful
  - E.g., randomly mutate well-formed inputs and runs the target application with the "*mutations*"



Fuzzing is great



In the best case, malformed inputs will explore different program paths, and trigger security vulnerabilities

However…

A quick example

```
1 void decode_image(FILE* fd){
2 ...
3 int length = get_length(fd);
4 int recomputed_chksum = checksum(fd, length);
5 int chksum_in_file = get_checksum(fd);
//line 6 is used to check the integrity of inputs
6 if(chksum_in_file != recomputed_chksum)
7   error();
8 int Width = get_width(fd);
9 int Height = get_height(fd);
10 int size = Width*Height*sizeof(int);//integer overflow
11 int* p = malloc(size);
12 ...
```

re-compute a new checksum

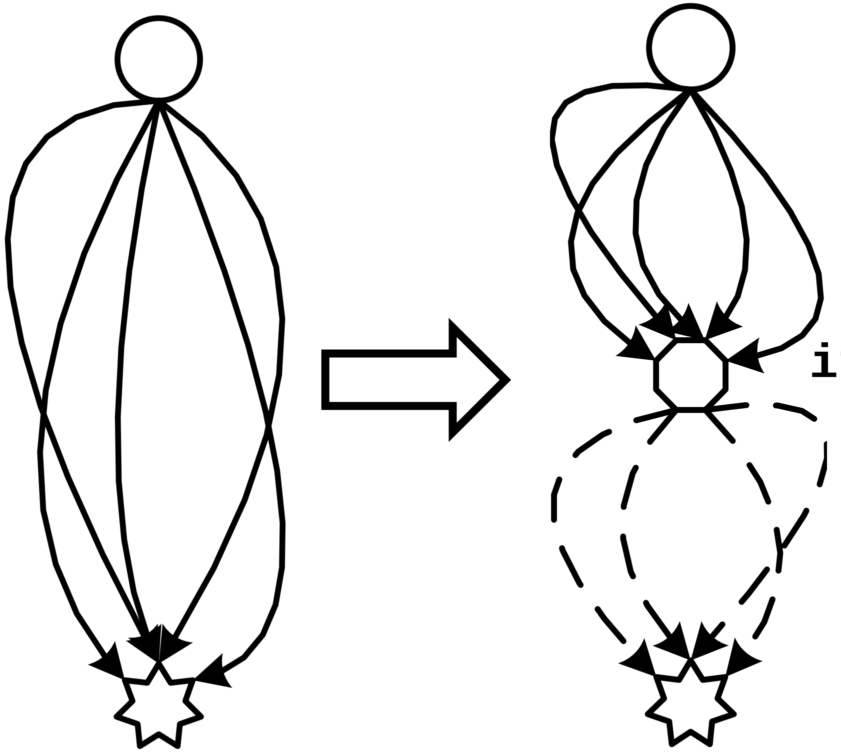
read the attached checksum

compare tow values

- **Malformed images will be dropped when the decoder function detects checksums mismatch**

Checksum: the bottleneck

Checksum is a common way to test the integrity of input data

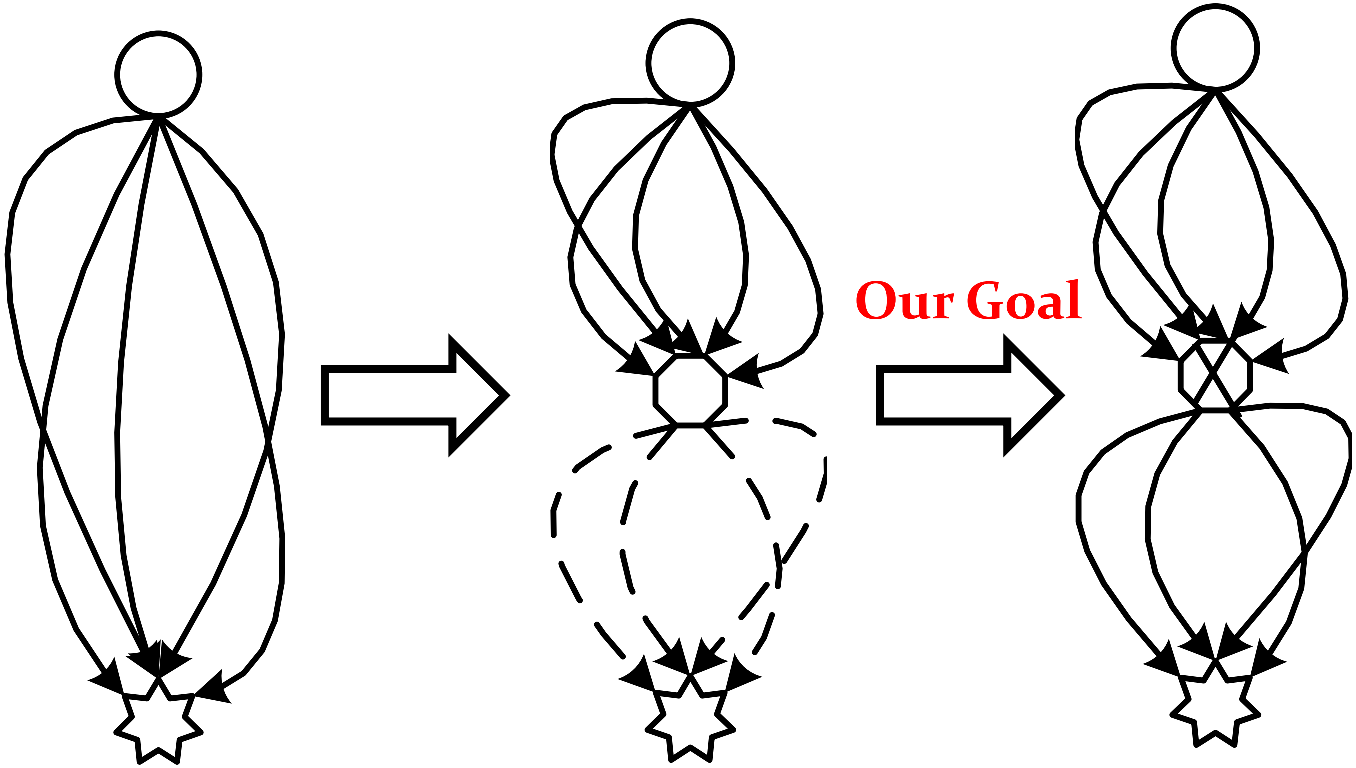


Most mutations are blocked at the checksum test point

`if(checksum(Data)!= Chksum)`

Our motivation

- **Penetrate checksum checks!**



Intuition

- **Disable checksum checks by control flow alteration**

```
if(checksum(Data)!= Chksum)
   goto L1;
   exit();
L1:
continue();
```

Original program
Modified program

- **Fuzz the *modified* program**
- **Repair the checksum fields in malformed inputs that can crash the modified program**

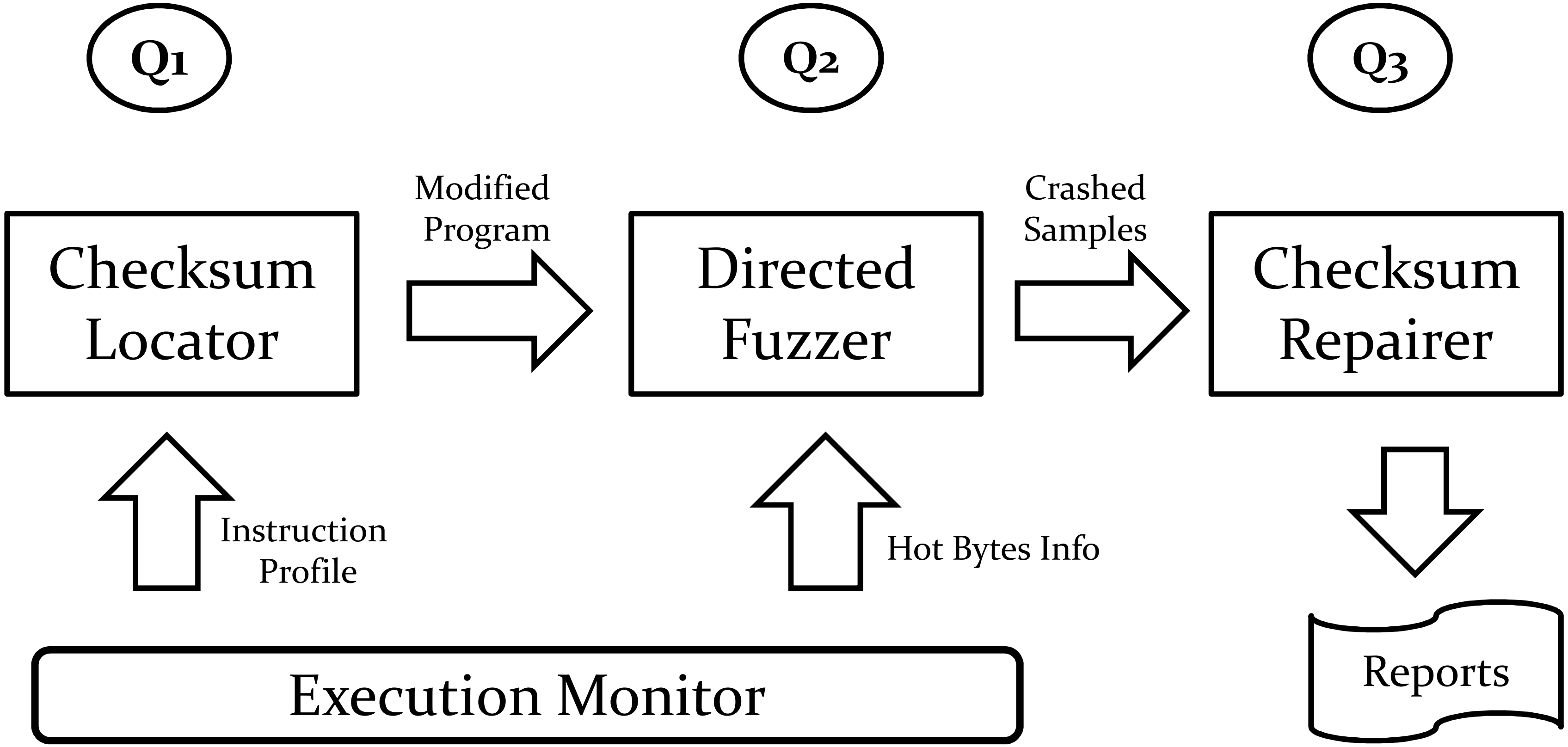
Key Questions

- **Q1: How to locate the checksum test instructions in a binary program?**

- **Q2: How to effectively and efficiently fuzz for security vulnerability detection?**

- **Q3: How to generate the correct checksum value for the invalid inputs that can crash the modified program?**

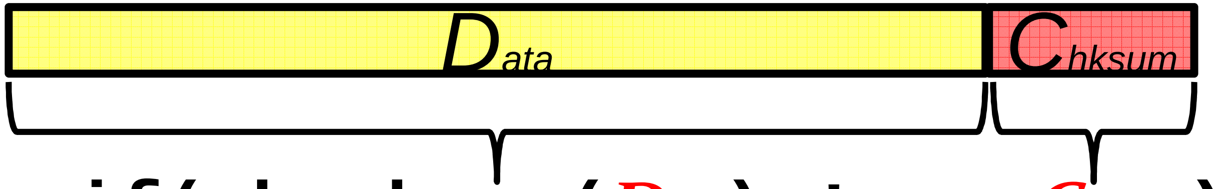
TaintScope Overview



A1: Locate the checksum test instruction

Key Observation 1

Checksum is usually used to protect a <u>large number</u> of input bytes



`if(checksum(Data) != Chksum)`

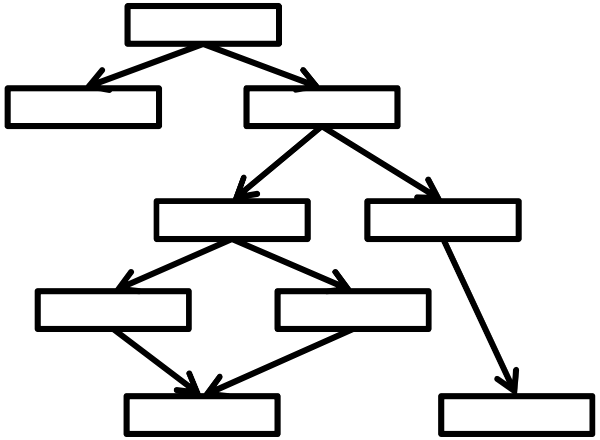
- Based on fine-grained taint analysis, we first find the <u>conditional jump instructions</u> (e.g., jz, je) that depend on more than a certain number of input bytes
- Take these conditional jump instructions as candidates

A1: Locate the checksum test instruction

Key Observation 2

Well-formed inputs can pass the checksum test, but most malformed inputs cannot

- We log the behaviors of candidate conditional jump instructions



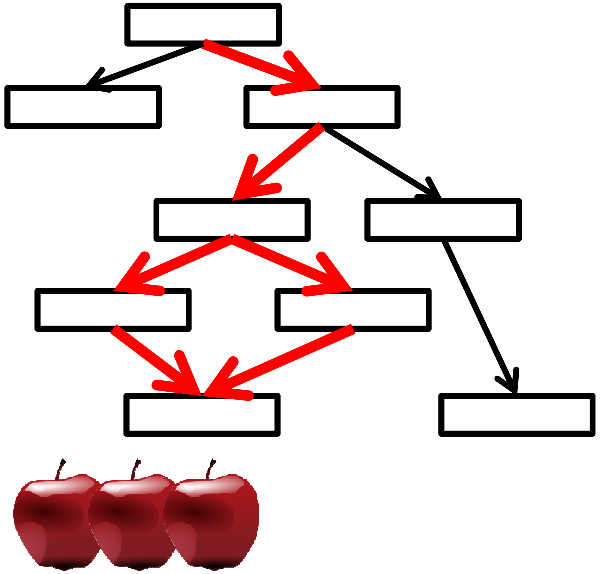
A1: Locate the checksum test instruction

Key Observation 2

Well-formed inputs can pass the checksum test, but most malformed inputs cannot

- We log the behaviors of candidate conditional jump instructions

① Run well-formed inputs, identify the always-taken and always-not-taken insts



A1: Locate the checksum test instruction

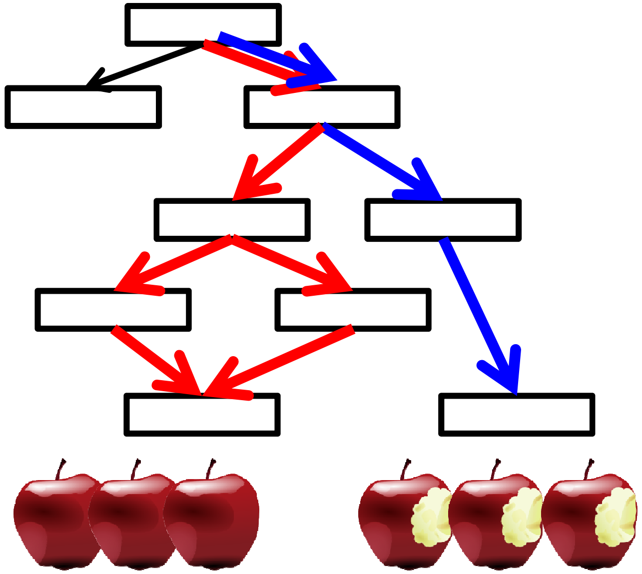
Key Observation 2

Well-formed inputs can pass the checksum test, but most malformed inputs cannot

- We log the behaviors of candidate conditional jump instructions

① Run well-formed inputs, identify the always-taken and always-not-taken insts

② Run malformed inputs, also identify the always-taken and always-not-taken insts



A1: Locate the checksum test instruction

Key Observation 2

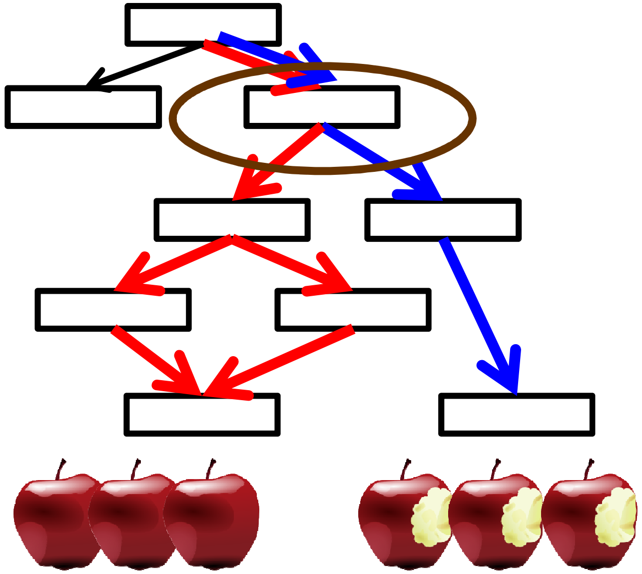
Well-formed inputs can pass the checksum test, but most malformed inputs cannot

- We log the behaviors of candidate conditional jump instructions

① Run well-formed inputs, identify the always-taken and always-not-taken insts

② Run malformed inputs, also identify the always-taken and always-not-taken insts

③ Identify the conditional jump inst that behaves completely different when processing well-formed and malformed inputs



A2: Effective and efficient fuzzing

- Blindly mutating will create huge amount of redundant test cases --- *ineffective and inefficient*

```
1 void decode_image(FILE
2 ...
  ...
6 if(chksum_in_file != recomput
    goto 8;
7   error();
8 int Width = get_width(fd);
9 int Height = get_height(fd);
10 int size = Width*Height*sizeof(int);//integer overflow
11 int* p = malloc(size);
12 …
```

Directly modifying "width" or "height" fields will trigger the bug easily

- Directed fuzzing: focus on modifying the "**hot bytes**" that refer to the input bytes flow into critical system/library calls
    - Memory allocation, string operation…

A3: Generate the correct checksum

- **The classical solution is symbolic execution and constraint solving**

Solving **`checksum(`**D_{ata}**`)==`** C_{hksum} is hard or impossible, if both D_{ata} and C_{hksum} are symbolic values

- **We use combined concrete/symbolic execution**
  - Only leave the bytes in the checksum field as symbolic values
  - Collect and solve the trace constraints on C_{hksum} when reaching the checksum test inst.
  - Note that:
    - **`checksum(`**D_{ata}**`)`** is a runtime determinable constant value.
    - C_{hksum} originates from the checksum field, but may be transformed, such as from hex/oct to dec number, from little-endian to big-endian.

Design Summary

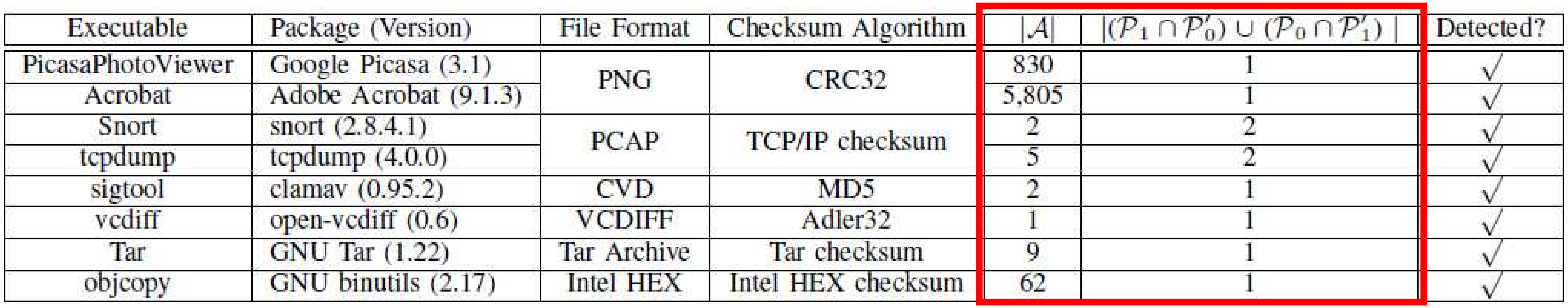
- **Directed Fuzzing**
  - Identify and modify "hot bytes" in valid inputs to generate malformed inputs
    - On top of *PIN* binary instrumentation platform
- **Checksum-aware Fuzzing**
  - Locate checksum check points and checksum fields.
  - Modify the program to accept all kinds input data
  - Generate correct checksum fields for malformed inputs that can crash the modified program
    - Offline symbolically execute the trace, using *STP* solver

Evaluation

- **Component evaluation**
  - E1: Whether TaintScope can locate checksum points and checksum fields?
  - E2: How many hot byte in a valid input?
  - E3: Whether TaintScope can generate a correct checksum field?
- **Overall evaluation**
  - E4: Whether TaintScope can detect previous unknown vulnerabilities in real-world applications?

Evaluation 1: locate checksum points

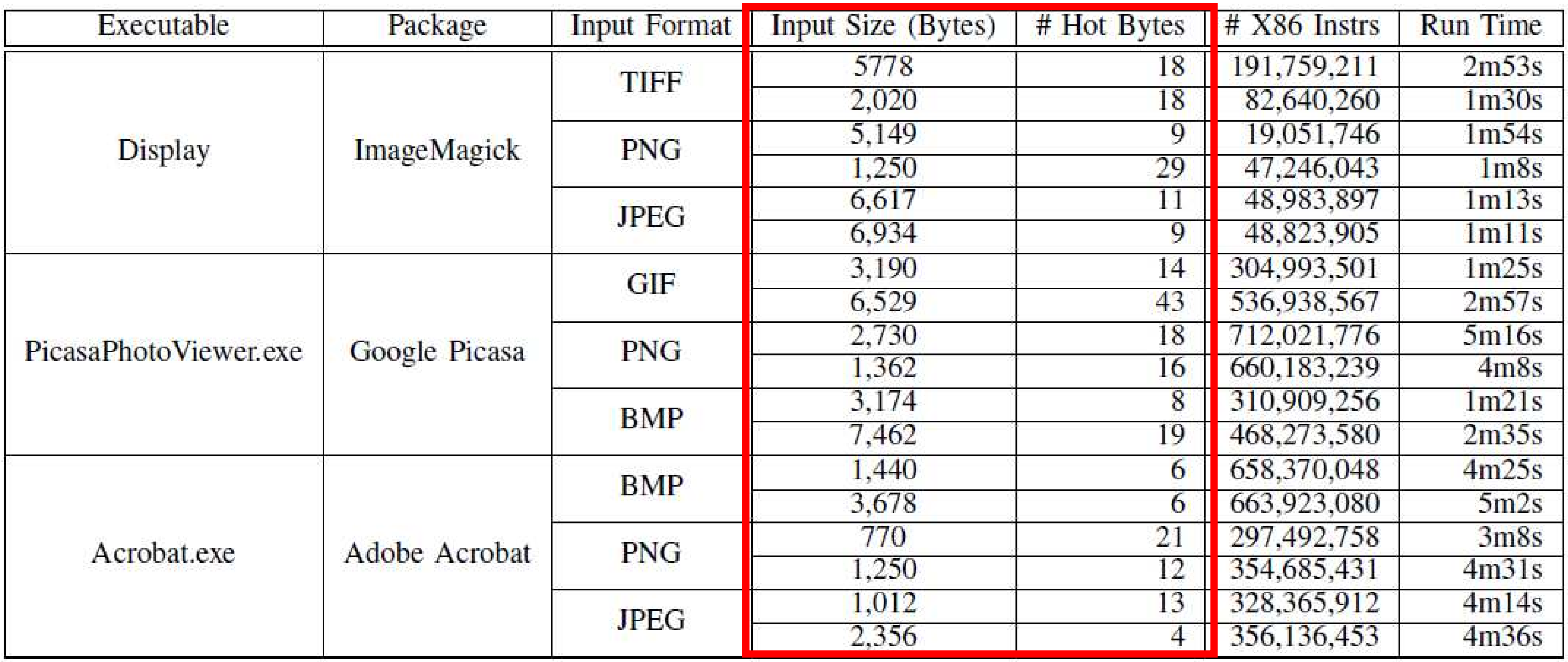
- We test several common checksum algorithms, including CRC32, MD5, Adler32. TaintScope accurately located the check statements.



Executable	Package (Version)	File Format	Checksum Algorithm	$\|\mathcal{A}\|$	$\|(\mathcal{P}_1 \cap \mathcal{P}'_0) \cup (\mathcal{P}_0 \cap \mathcal{P}'_1)\|$	Detected?
PicasaPhotoViewer	Google Picasa (3.1)	PNG	CRC32	830	1	√
Acrobat	Adobe Acrobat (9.1.3)			5,805	1	√
Snort	snort (2.8.4.1)	PCAP	TCP/IP checksum	2	2	√
tcpdump	tcpdump (4.0.0)			5	2	√
sigtool	clamav (0.95.2)	CVD	MD5	2	1	√
vcdiff	open-vcdiff (0.6)	VCDIFF	Adler32	1	1	√
Tar	GNU Tar (1.22)	Tar Archive	Tar checksum	9	1	√
objcopy	GNU binutils (2.17)	Intel HEX	Intel HEX checksum	62	1	√

Evaluation 2: identify hot bytes

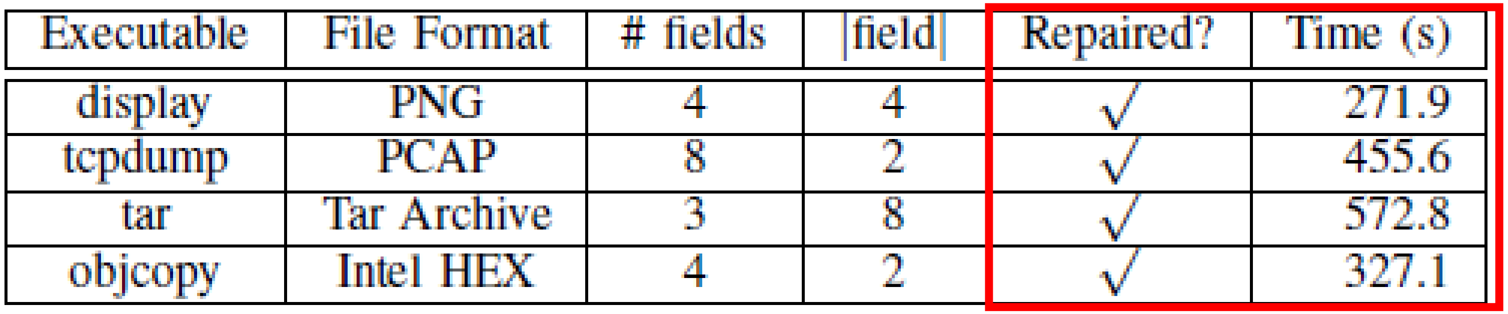
- We measured the number of bytes could affect the size arguments in memory allocation functions



Executable	Package	Input Format	Input Size (Bytes)	# Hot Bytes	# X86 Instrs	Run Time
Display	ImageMagick	TIFF	5778	18	191,759,211	2m53s
			2,020	18	82,640,260	1m30s
		PNG	5,149	9	19,051,746	1m54s
			1,250	29	47,246,043	1m8s
		JPEG	6,617	11	48,983,897	1m13s
			6,934	9	48,823,905	1m11s
PicasaPhotoViewer.exe	Google Picasa	GIF	3,190	14	304,993,501	1m25s
			6,529	43	536,938,567	2m57s
		PNG	2,730	18	712,021,776	5m16s
			1,362	16	660,183,239	4m8s
		BMP	3,174	8	310,909,256	1m21s
			7,462	19	468,273,580	2m35s
Acrobat.exe	Adobe Acrobat	BMP	1,440	6	658,370,048	4m25s
			3,678	6	663,923,080	5m2s
		PNG	770	21	297,492,758	3m8s
			1,250	12	354,685,431	4m31s
		JPEG	1,012	13	328,365,912	4m14s
			2,356	4	356,136,453	4m36s

Evaluation 3: generate correct checksum fields

- **We test malformed inputs in four kinds of file formats.**
- **TaintScope is able to generate correct checksum fields.**



Executable	File Format	# fields	|field|	Repaired?	Time (s)
display	PNG	4	4	√	271.9
tcpdump	PCAP	8	2	√	455.6
tar	Tar Archive	3	8	√	572.8
objcopy	Intel HEX	4	2	√	327.1

Evaluation 4 : 27 previous unknown vulns

MS Paint

Google Picasa

Adobe Acrobat

ImageMagick

irfanview

gstreamer

Winamp

XEmacs

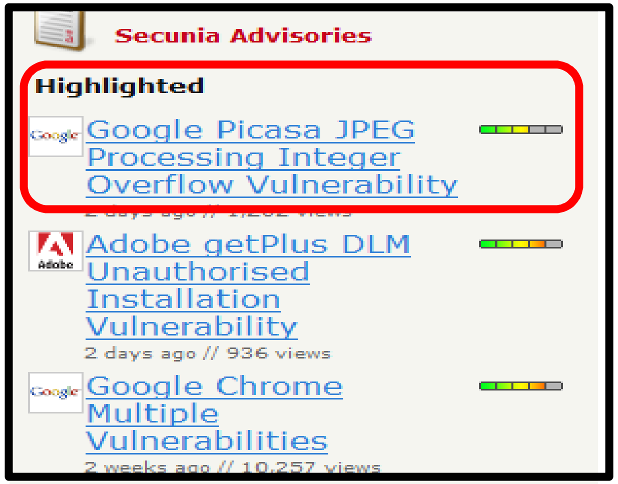
Amaya

dillo

wxWidgets

PDFlib

Evaluation 4 : 27 previous unknown vulns



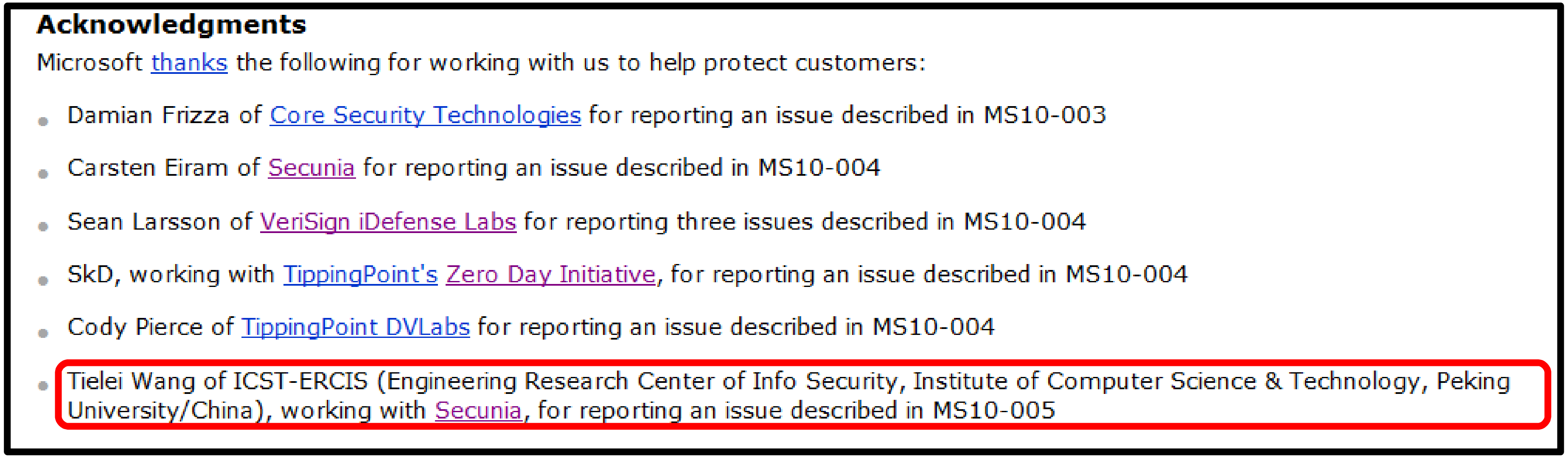
Secunia Advisories

Highlighted

- Google Picasa JPEG Processing Integer Overflow Vulnerability
  2 days ago // 1,262 views
- Adobe getPlus DLM Unauthorised Installation Vulnerability
  2 days ago // 936 views
- Google Chrome Multiple Vulnerabilities
  2 weeks ago // 10,257 views

Adobe would like to thank the following individuals and organizations for reporting the relevant issues and for working with Adobe to help protect our customers' security:

- Michael Schmidt of Compass Security (http://www.csnc.ch) (CVE-2007-0048, CVE-2007-0045)
- Didier Stevens (CVE-2009-2979)
- Drew Yao of Apple Product Security (http://www.apple.com/support/security/) (CVE-2009-2980)
- Stefano Di Paola of Minded Security (http://www.mindedsecurity.com/) (CVE-2009-2981)
- Guillaume Delugré and Frédéric Raynal of SOGETI ESEC (http://esec.fr.sogeti.com/) (CVE-2009-2982, CVE-2009-3461, CVE-2009-3462)
- SkyLined of Google Inc. (http://skypher.com/SkyLined) (CVE-2009-2983)
- Tavis Ormandy, Google Security Team (http://www.google.com/corporate/security.html) (CVE-2009-2984)
- An anonymous researcher reported through TippingPoint's Zero Day Initiative (http://www.zerodayinitiative.com/) (CVE-2009-2985)
- Will Dormann, CERT (http://www.cert.org/) (CVE-2009-2986)
- Zhenhua Liu and Xiaopeng Zhang of Fortinet's FortiGuard Global Security Research Team (http://www.fortiguardcenter.com) (CVE-2009-2987, CVE-2009-2988, CVE-2009-2996)
- Tielei Wang from ICST-ERCIS (Engineering Research Center of Info Security, Institute of Computer Science & Technology, Peking University / China) (CVE-2009-2989, CVE-2009-2995)

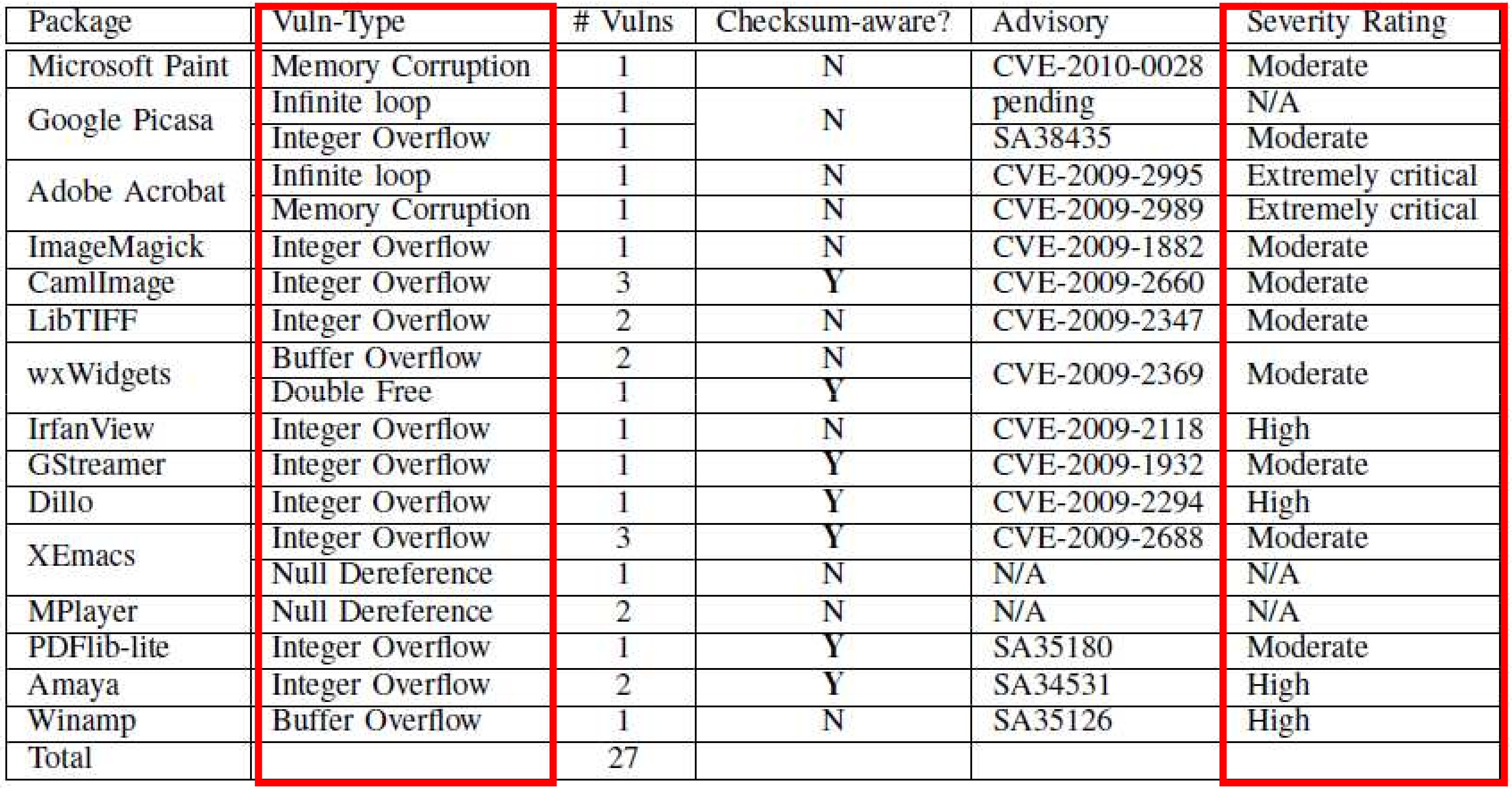


Acknowledgments

Microsoft thanks the following for working with us to help protect customers:

- Damian Frizza of Core Security Technologies for reporting an issue described in MS10-003
- Carsten Eiram of Secunia for reporting an issue described in MS10-004
- Sean Larsson of VeriSign iDefense Labs for reporting three issues described in MS10-004
- SkD, working with TippingPoint's Zero Day Initiative, for reporting an issue described in MS10-004
- Cody Pierce of TippingPoint DVLabs for reporting an issue described in MS10-004
- Tielei Wang of ICST-ERCIS (Engineering Research Center of Info Security, Institute of Computer Science & Technology, Peking University/China), working with Secunia, for reporting an issue described in MS10-005

Evaluation 4: 27 previous unknown vulns



Package	Vuln-Type	# Vulns	Checksum-aware?	Advisory	Severity Rating
Microsoft Paint	Memory Corruption	1	N	CVE-2010-0028	Moderate
Google Picasa	Infinite loop	1	N	pending	N/A
	Integer Overflow	1		SA38435	Moderate
Adobe Acrobat	Infinite loop	1	N	CVE-2009-2995	Extremely critical
	Memory Corruption	1	N	CVE-2009-2989	Extremely critical
ImageMagick	Integer Overflow	1	N	CVE-2009-1882	Moderate
CamlImage	Integer Overflow	3	Y	CVE-2009-2660	Moderate
LibTIFF	Integer Overflow	2	N	CVE-2009-2347	Moderate
wxWidgets	Buffer Overflow	2	N	CVE-2009-2369	Moderate
	Double Free	1	Y		
IrfanView	Integer Overflow	1	N	CVE-2009-2118	High
GStreamer	Integer Overflow	1	Y	CVE-2009-1932	Moderate
Dillo	Integer Overflow	1	Y	CVE-2009-2294	High
XEmacs	Integer Overflow	3	Y	CVE-2009-2688	Moderate
	Null Dereference	1	N	N/A	N/A
MPlayer	Null Dereference	2	N	N/A	N/A
PDFlib-lite	Integer Overflow	1	Y	SA35180	Moderate
Amaya	Integer Overflow	2	Y	SA34531	High
Winamp	Buffer Overflow	1	N	SA35126	High
Total		27			

Conclusion

- **Checksum is a big challenge for fuzzing tools**
- **TaintScope can perform:**
  - Directed fuzzing
    - Identify which bytes flow into system/library calls.
    - dramatically reduce the mutation space.
  - Checksum-aware fuzzing
    - Disable checksum checks by control flow alternation.
    - Generate correct checksum fields in invalid inputs.
- **TaintScope detected dozens of serious previous unknown vulnerabilities.**

Thanks for your attention!