

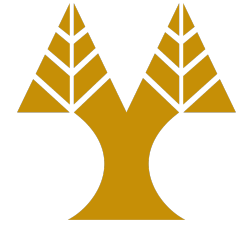
CS451 – Software Analysis

Lecture 14

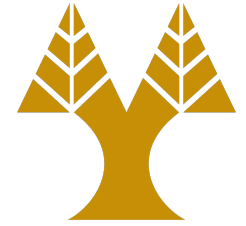
Dynamic Taint Analysis

Elias Athanasopoulos
elathan@cs.ucy.ac.cy

Dynamic Taint Analysis (DTA)



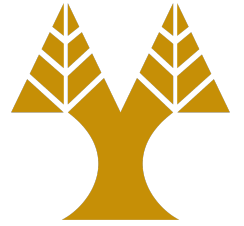
- DTA is an analysis for determining the influence of a specific program state to other parts of the program state
 - Imagine that you can mark all data coming from the network
 - These marked bytes will be processed by the program logic, and new data that are somehow based on the marked bytes, will be also marked
 - If the program counter attempts to execute marked data, then raise an alert
- This marking is called *tainting*
- DTA can be also called data-flow tracking (DFT), taint tracking, or taint analysis



DTA on binaries

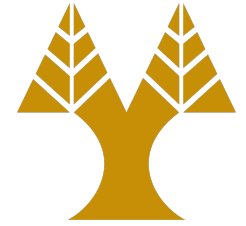
- Implemented over a DBI framework, such as Pin
- DTA instruments all instructions that handle data (registers or memory)
 - Most instructions can influence memory, so DTA instruments almost every instruction leading to high overheads
 - DTA is applied on off-line analysis and not on production code
- DTA can be also applied statically when source code is available
 - The compiler emits the instrumentation

Sources, sinks and propagation



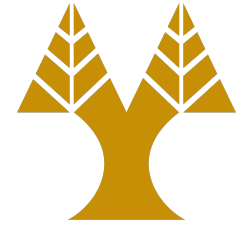
- For a DTA we need to define what is an interesting state, and how data influence other data
- DTA can be used to solve different problems so what is interesting in each case may be very different
- For a DTA, we need to define three elements
 - Taint sources (defined by the analyst)
 - Taint sinks (defined by the analyst)
 - Taint propagation (implemented by the engine, but can be customizable)

Taint sources



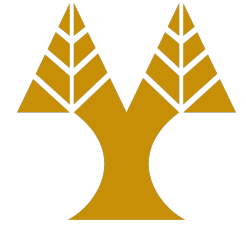
- Program locations where we *taint* the data that is interesting for the analysis
 - arguments that are passed in system calls
 - arguments that are passed in specific functions
 - data that is the output of a `read()` call
- The DTA engine allows you to declare which data are going to be tainted

Taint sinks



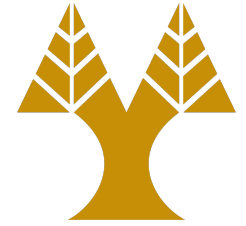
- Locations in the program that can be influenced by tainted data
 - Consider indirect jumps, which use memory to direct the control flow of the program
 - An analysis can declare all indirect call/jumps as sinks
 - The DTA engine will instrument all such calls/jumps and infer if the values used (register, memory) are tainted

Taint propagation



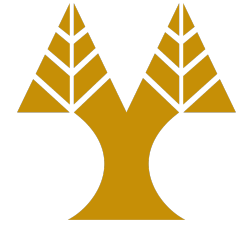
- Tainted data is processed with other data, and taint can flow from already tainted data to untainted data
 - Consider a `mov` instruction that copies a tainted value to a new memory location
 - The new location now is also tainted
- Tracking taint is complicated and needs instrumentation in the majority of the program's instructions
- Different taint policies dictate different taint propagation rules

DTA and the Heartbleed bug



- Heartbleed is a *buffer overread* bug
 - Allows any client to exfiltrate sensitive data from a web server by crafting a very specific request
 - The bug is in the OpenSSL library, which is used for cryptographic operations
 - Many web servers use OpenSSL to implement TLS
- Heartbleed exploits a bad implementation of the Heartbeat protocol
 - A client sends a special request with a string and its length to a server, which should be echoed back to the client
 - A buggy implementation of the Heartbeat protocol, allows an attacker to insert an arbitrary length in the request
 - A large length value coerces the server to copy much more than it is needed for the reply (overread)

Heartbleed code



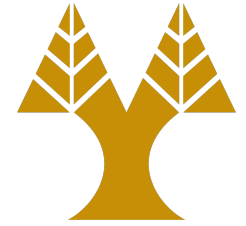
```
buffer = OPENSSL_malloc(1 + 2 + payload + padding);
bp = buffer;

*bp++ = TLS1_HB_RESPONSE;
s2n(payload, bp);

/* pl, payload are both attacker controlled */
memcpy(bp, pl, payload);
bp += payload;

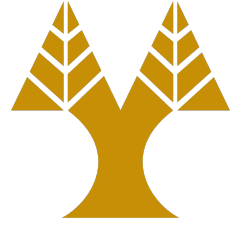
RAND_pseudo_bytes(bp, padding);
r = ssl3_write_bytes(s,
    TLS1_RT_HEARTBEAT,
    buffer, 3 + payload + padding);
```

How bad can it be?



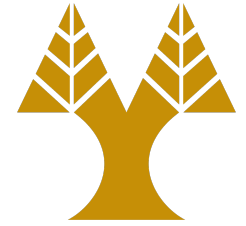
- Heartbleed coerces the server to copy a large buffer to a network buffer that is sent to the attacker
- This is not a buffer overflow bug, but an overread
 - The destination buffer is big enough to hold the data
 - The source buffer can be very small, and the copy will eventually read other data close to the source buffer
 - If there is a sensitive cryptographic key (private key) then the key is copied to the network buffer

Detecting Heartbleed with DTA



- Taint sources
 - Sensitive data in memory (e.g., a private key)
- Taint sinks
 - `send( )` and `sendto( )`

In action



pl

f	o	o	b	a	r
?	?	?	?	?	?
s	e	c	r	e	t
k	e	y			

strlen(pl) = 6, payload = 21

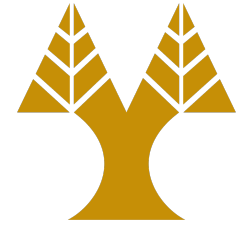
bp

f	o	o	b	a	r
?	?	?	?	?	?
s	e	c	r	e	t
k	e	y			

memcpy

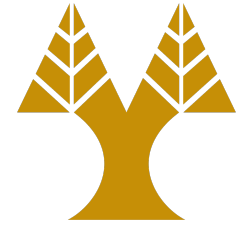
f	o	o	b	a	r
?	?	?	?	?	?
s	e	c	r	e	t
k	e	y			

DTA Design



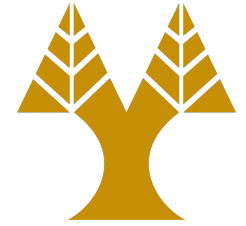
- Granularity
 - The unit of information that taint is applied (bit, byte, word, etc.)
- Colors
 - Taint may have different levels of marking
- Policies
 - How taint propagates when data is part of an expression

Taint granularity



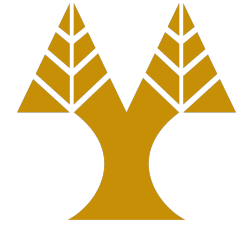
- Taint can be applied to different levels of information
- Bit-level example (red is tainted)
00101101 & 00000100 = 00000**1**00
 - If the attacker controls the entire byte the only bit that can change is the tainted one
- Byte-level example (red is tainted)
00101101 & 00000100 = **00000100**
 - The system considers that the attacker can affect all computation, which is not true
- Important trade-off
 - Tracking taint at the bit level is more accurate, but more expensive

Taint colors



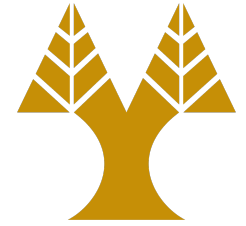
- Some DTA applications may need to differentiate tainted data originating from taint sources
 - Taint sources may use a different taint color to mark data
 - Taint sinks may conclude in different decisions based on the taint color
- Colors require the DTA to store more information per byte (not just a bit for taint/no taint)
- A byte could store 256 different colors, however, colors can be *mixed*
 - Data from different sources may contribute to an expression

Taint colors example



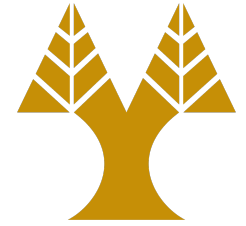
- Assume that we have 1 byte for storing taint information
 - We can support 8 colors: 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80
- If a data value is tainted by both 0x01 and 0x02, then we can use bitwise OR to derive a new color
 - $0x01 \text{ OR } 0x02 = 0x03$

Taint propagation policies



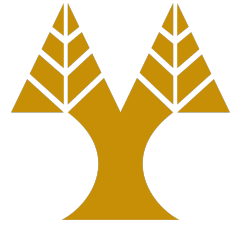
- Tainted values participate and contribute to expressions
- The way taint propagates, when data is processed, defines the DTA engine's policies
- For the following example we assume a byte-level DTA engine that supports two colors "red" (R) and "blue" (B), and we assume expressions that support 4-byte operands (typical for 32-bit architectures)

Example of taint propagation



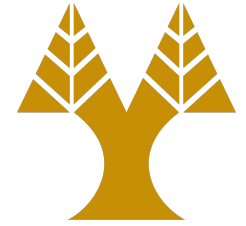
Operation	x86	a	b	c	Op
$c = a$	mov	[R][B][R][B]		[R][B][R][B]	:=
$c = a \text{ xor } b$	xor	[R][][][R]	[B][RB][B][RB]	[RB][RB][B][RB]	U
$c = a + b$	add	[R][R][][R]	[][][B][B]	[R][R][B][RB]	U
$c = a \text{ xor } a$	xor	[R][RB][B][RB]		[][][][]	∅

Overtainting and undertainting



- Depending on the policy, the DTA engine may suffer from overtainting or undertainting
- Undertainting
 - Values that should be tainted, are not
 - For instance, some CPU instructions may not be instrumented for propagating taint
- Overtainting
 - Values end up tainted, although they should not
 - This can lead to false positives

Control dependencies

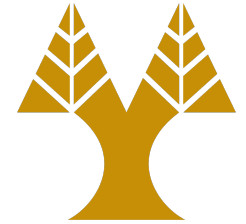


- Memory can influence other memory, *implicitly*
 - In that case, taint is not propagated
- Example of implicit flow

```
var = 0;
while (cond--) var++;
```

An attacker that controls `cond` can influence `var`, but the two variables do not directly interact
- One solution is to propagate tainting in loops, but this can lead to massive overtainting

Shadow memory



Virtual Memory			
DE	8A	42	1F
A	B	C	D

1	1	0	1
A	B	C	D

1 shadow bit/byte
(1 color)

01	04	0	20
A	B	C	D

1 shadow byte/byte
(8 colors)

A	01	00	00	00
B	01	00	00	00
C	00	80	00	00
D	00	00	02	00

4 shadow byte/byte
(32 colors)