Decker: Attack Surface Reduction via On-Demand Code Mapping

Chris Porter, Sharjeel Khan, Santosh Pande

ASPLOS 2023
Problem: Code reuse attacks

- Software continues to be susceptible to existing and new code reuse attacks
- **Code reuse attacks** are software attacks that leverage existing code in programs to perform some malicious action
- They’re commonly built today with **gadgets**
- **Gadgets** are code snippets that can be stitched together at runtime to form **gadget chains** that execute malicious behavior
ROP chain example

ROP gadgets within the .text section

```
0x500000
...  
0x52e1fe
...  
0x55720a
...  
0x600000

.text

0x5001f9
...  
pop rcx
ret

mov qword ptr [rcx], rax
ret

pop rax
ret
```
### ROP chain example

**Stack payload**

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x55720a</td>
<td>0x2a</td>
</tr>
<tr>
<td>0x5001f9</td>
<td>0x800000</td>
</tr>
<tr>
<td>0x52e1fe</td>
<td>0x55720a</td>
</tr>
</tbody>
</table>

```
Stack payload
```

```
.text

    0x500000
    ...      
    0x5001f9
    ...      
    0x52e1fe
    ...      
    0x600000

    pop rcx
    ret

    mov qword ptr [rcx],
    rax
    ret

    pop rax
    ret
```

**.text**
ROP chain example

Stack payload

0x55720a
0x2a
0x5001f9
0x800000
0x52e1fe
0x55720a
0x600000

.gadget 1 address -> pop rax; ret;

.text

0x500000
... 
0x5001f9
... 
0x52e1fe
... 
0x600000

pop rcx
ret
mov qword ptr [rcx], rax
ret
pop rax
ret
ROP chain example

```
.text

0x500000
...  
0x5001f9
...  
0x52e1fe
...  
0x55720a
...  
0x600000

pop rcx
ret
mov qword ptr [rcx], rax
ret
pop rax
ret
```

Stack payload

0x55720a
0x2a
0x5001f9
0x800000
0x52e1fe
0x55720a
ROP chain example

Stack payload

0x55720a 0x2a 0x5001f9 0x800000 0x52e1fe
0x500000 ...
0x50001f9 ...
0x52e1fe ...
0x600000 ...

.text

pop rcx
ret
mov qword ptr [rcx], rax
ret
pop rax
ret
ROP chain example

Stack payload

0x55720a
0x2a
0x5001f9
0x800000
0x52e1fe
0x55720a
0x600000

.text

0x500000

... 0x5001f9
... 0x52e1fe
... 0x55720a
... 0x600000

pop rcx
ret
mov qword ptr [rcx], rax
ret
pop rax
ret
ROP chain example

Stack payload

0x55720a
0x2a
0x5001f9
0x800000
0x52e1fe
0x55720a
0x52e1fe
0x500000
0x600000

.text

0x500000
... 0x5001f9
... 0x52e1fe
... 0x55720a
... 0x55720a
... 0x600000

pop rcx
ret
mov qword ptr [rcx], rax
ret
pop rax
ret
ROP chain example

 Stack payload

0x55720a
0x2a
0x5001f9
0x800000
0x52e1fe
0x55720a

.text

0x500000
... 0x5001f9
... 0x52e1fe
... 0x55720a

pop rcx
ret
mov qword ptr [rcx],
xax
ret
pop rax
ret

0x600000
ROP chain example

Stack payload

0x55720a
0x2a
0x5001f9
0x800000
0x52e1fe
0x55720a
0x600000

.text

0x500000
...
ROP chain example

- Summary: gadget chaining works by leveraging multiple snippets across the existing code.
Debloating as defense

- We know that software is bloated with unused code
  - The unneeded code contains gadgets which could be chained.
  - Why not remove it?
- Debloating is a proposed defense against code reuse attacks
- Shortcomings with current approaches
  - Too conservative, leaving too much code available to attackers
  - Or they compromise soundness by specializing the application for only certain inputs or features, which can lead to crashes or incorrect output
Soundness and may-use code

**Definition**

**sound transformation**: a program transformation that does not change the semantics of a program. Program transformations that induce crashes or cause incorrect output are unsound.

**Definition**

**may-use code**: code that may be used by the program under certain inputs or execution conditions.
## Properties of debloating frameworks

<table>
<thead>
<tr>
<th></th>
<th>Piece-wise</th>
<th>Chisel</th>
<th>Razor</th>
<th>BlankIt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Works on application</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td></td>
</tr>
<tr>
<td>Works on library</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>Works on binary</td>
<td></td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>No user input needed</td>
<td>✔️</td>
<td></td>
<td></td>
<td>✔️</td>
</tr>
<tr>
<td>No training needed</td>
<td>✔️</td>
<td>✔️</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Is sound</td>
<td>✔️</td>
<td></td>
<td></td>
<td>✔️</td>
</tr>
<tr>
<td>Can debloat may-use code</td>
<td></td>
<td></td>
<td>✔️</td>
<td>✔️</td>
</tr>
</tbody>
</table>
## Properties of debloating frameworks

<table>
<thead>
<tr>
<th></th>
<th>Piece-wise</th>
<th>Chisel</th>
<th>Razor</th>
<th>Blanklt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Works on application</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td></td>
</tr>
<tr>
<td>Works on library</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>Works on binary</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>No user input needed</td>
<td>✔️</td>
<td>✔️</td>
<td></td>
<td>✔️</td>
</tr>
<tr>
<td>No training needed</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>Is sound</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>Can debloat may-use code</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
</tbody>
</table>
### Properties of debloating frameworks

<table>
<thead>
<tr>
<th></th>
<th>Piece-wise</th>
<th>Chisel</th>
<th>Razor</th>
<th>BlankIt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Works on application</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>X</td>
</tr>
<tr>
<td>Works on library</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Works on binary</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>No user input needed</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>No training needed</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Is sound</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Can debloat may-use code</td>
<td></td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
</tbody>
</table>
Motivation

To the best of our knowledge, there is no general technique today that:

1. Works on the applications as a whole instead of libraries
2. Is sound
3. Can effectively debloat may-use code using dynamic contexts
Outline

1. Introduction
2. Overview
3. Decking
4. Evaluation
5. Conclusion
Decker overview

- Decker is a **constructive** attack surface reduction technique
- “Decks” are active sections of code that the program can effectively stand on. When a deck is unneeded, it can be removed.
- Map only code that is currently needed by a running program
- Disable all other code so accesses trigger a runtime exception
- Granularity: implement with system code pages (4KB)
- Compiler and runtime component to it
Threat model

- **Focus is on attack surface reduction**
  - Assume the attacker can initiate and propagate the attack
  - Decker’s main goal
    - Incrementally expose the executable surface of a program by following its interprocedural control flow.
    - Breaks chains of gadgets, because all the gadgets that compose a chain are never dynamically exposed at the same time.

- **Integrity of indirect call targets is out of scope** (orthogonal schemes like CFI and CPI are designed specifically to tackle it)
A deck

Image source: Wikipedia (deck)
Decker Example

main()

foo()

bar()
Decker Example

```
main()
...  
if(x)
  foo()
  bar()
foo()
...  
bar()
...  
```
Decker Example

```
main()
...
if(x)
  foo()
  bar()

foo()
...

bar()
...
```
Decker Example

main()
... 
if(x)
  foo()
  bar()

foo()
  ...

bar()
  ...

Decker Example

main()

... 
if(x)
  foo()
  bar()

foo()
...

bar()
...
Decker Example

```
main()
...
if(x)
    foo()
    bar()
foo()
...
bar()
...
```
Decker Example

```
main()
...
if(x)
foo()
bar()
foo()
...
bar()
...
```
Decker Example

```
main()
...
if (x)
  foo()
  bar()

foo()
...
bar()
...
```
Decker Example

```
main()
...
if (x)
  foo()
  bar()
foo()
...
bar()
...
```
Decker Example

```
main()
...
if(x)
  foo()
  bar()
foo()
...
bar()
...
```
Example (Unprotected Program)

Gadget chain components: 🐜 🐜 🐜

Gadget chain possible? ✔
Example (Decker-Protected Program)

Gadget chain components:

Gadget chain possible?
Example (Decker-Protected Program)

Gadget chain components:
- main()
- foo()
- bar()

Gadget chain possible? No
Example (Decker-Protected Program)

Gadget chain components:

foo() bar()

Gadget chain possible?

✗ ✓
Outline

1. Introduction
2. Overview
3. Decking
4. Evaluation
5. Conclusion
Program structure and performance

- Ideally, decking should be done at the entrances and exits of callsites
  - But such instrumentation can cause substantial runtime overhead
- *Loops* in particular are problematic
  - Instrumentation for functions invoked inside of loops will execute repeatedly and kill performance
- Leads us to 4 types of decks
Example

main()
A()
B()
while {
    B()
} A(func_ptr)
B()
func_ptr()
C()

C()
Example

```
main()
A(func_ptr)
while {
    B()
}
A(func_ptr)
B()
func_ptr()
B()
C()
```
main()
A(func_ptr)
while {
  B()
}
A(func_ptr)
B()
func_ptr()
B()
C()
Example

main()

A(func_ptr)
while {
    B()
}

A(func_ptr)
    B()
    func_ptr()

B()
C()
Let’s discuss the 4 deck types!

```
main()
  A(func_ptr)
  while {
    B()
    B()
    func_ptr()
    B()
    C()
  }
  A(func_ptr)
```
1. Single deck - Occurs in a non-loop region

```c
#include <stdio.h>

int main() {
    int RX = 100;
    map(RX, A);

    A(func_ptr);
    while {
        B();
    }
    A(func_ptr);
    B();
    func_ptr();
    B();
    C();
}
```
2. Loop deck - Entrance to loops

```cpp
main()
A()
B()
while {
    B()
}
A(func_ptr)
B()
func_ptr()
B()
C()
map(RX, B, C)
```
3. Reachable deck - Enter loop region via non-loop

```
main()
A(func_ptr)
while {
B()
}
A(func_ptr)
func_ptr()
B()
C()
```

map(RX, B, C)
4. Indirect deck - Occurs at indirect calls

main()
A(func_ptr)
while {
  B()
}
A(func_ptr)
func_ptr()
B()
Indirect deck

- Static function pointer analysis?
  - Narrows possible targets but is an overapproximation
  - Limits attack surface reduction
- Instead, resolve this at runtime by
  - Passing function pointer to Decker runtime
  - Mapping the appropriate pages
- We will violate the rule of thumb and instrument inside of loops!
  - Caching optimization needed
Problem: Functions in one deck can occupy the same code page as functions in another deck, and we do not want to inadvertently include gadgets from other functions when we map a code page as active.

Solution: Separate deck sets into disjoint sets (separate pages) and use a linker script to enforce it.
Outline

1. Introduction
2. Overview
3. Decking
4. Evaluation
5. Conclusion
Evaluation goals

- What is the performance slowdown due to Decker?
- What is the gadget reduction for applications that use Decker?
- Can Decker break real gadget chains in the benchmarks and real-world applications to be able to stop gadget-based attacks?
- render JOP gadgets ineffective in practical scenarios, including Windows?
Evaluation summary

- **Performance**
  - ~5% average overhead for SPEC 2017, GNU coreutils, and nginx

- **Security**
  - 70-87% total gadget reduction
  - Equals or (in many cases) improves on comparable prior work
  - Achieves this without compromising soundness
Evaluation summary

- Gadget chain-breaking
  - An unexplored metric that we introduce and report for Decker
  - Decker fully breaks a popular syscall ROP chain in all cases
  - Decker removes functionality which is critical for JOP chains to be successful from nginx in both Linux and Windows
Evaluation details

- Please see paper for detailed results and analysis!
Conclusion

- Decker
  - an attack surface reduction technique for applications
  - is sound and enables may-use code on-demand
  - requires zero training, user inputs, or specifications

- Acceptable performance slowdowns and strong total gadget reductions across two benchmark suites and nginx

- Gadget chain-breaking study
  - Breaks automatically generated syscall ROP chains
  - Strong evidence that useful JOP chains are substantially hampered or impossible to build for both Linux and Windows
Thank you!

- Check out our artifact!
  - [https://zenodo.org/record/7319957](https://zenodo.org/record/7319957)
  - Link is available in the paper, too

- Questions?