True Code

Automated security vulnerability checks

riscure

POINTER : TIME OF CHECK – TIME OF USE

If a pointer is coming from an untrusted environment and an attacker can change the pointer value between the time of check and the time of use, this introduces a security vulnerability

- ✓ This check searches for pointers which are dereferenced more than once
- ✓ The check is most effective if the context that the check uses contains security sensitive code sections

Code example

```
void public_function(uint8_t* index, uint8_t value) {
        if (*index < array_size) {
        array[*index] = value;    //2nd dereference
        }
}
```

! The configuration of this check accepts a list of functions to exclude. This will reduce the number of false positives

FUNCTION ARGUMENT VALIDATION

The value of an argument in a function call may be out of range. This can cause overflows or other unexpected behavior that can be exploited by an attacker

- ✓ This code check searches for missing argument validation in function paths
- ✓ The configuration of the check allows to configure the start and end of a function path

Code example

```
void func_decl(int l, int k) {
  if (k > 0) {
            func(k, l);     //range l not checked in path
  }
}

void func(int a, int b){
   func_call(a, b);       //range l not checked in path
}
```

! All checks can be configured to run multiple times. Each time on a different set of context

INTEGER OVERFLOW VALIDATION

Integer multiplication may lead to integer overflow. This results in a vulnerability that can be exploited by an attacker

- ✓ The code check searches for integer or similar type function arguments
- ✓ For these arguments the check searches for appropriate range checks
- ✓ Multiplications and shift left are taken into account

Code example

```
int func_1(int k) {
        return k * 256; //int multiply without prior validation
}
```

! The results of any check are added to a database of your choice. All users of True Code immediately see the vulnerabilities marked in the code base they are working on

RETURN VALUE USAGE CHECK

Security functions usually have a return type that can be used by the caller to verify if the function ended normally and to decide how to proceed based on the outcome of the function. Not checking the return value of a security function can introduce unwanted behavior that can be exploited by an attacker.

- ✓ Context can be set for this check so that only functions important for the security of the application are validated
- ✓ A threshold can be set so that only exceptional coding patters trigger a warning

Code example

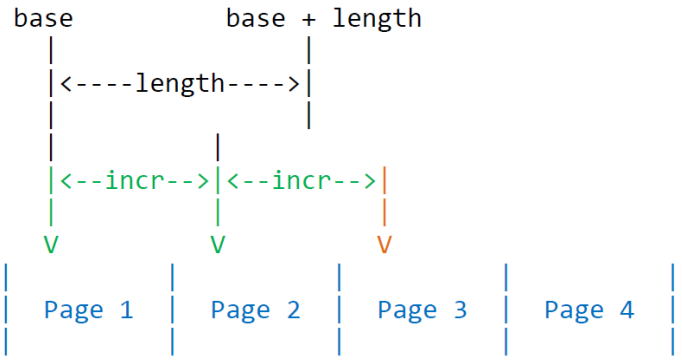
```
int func_1(int k) {
        return k + 2;
}

void func_2() {
        int l = func_1(1);      //33%
        int m = func_1(2);   //33%
        func_1(3);      //return not used against
66%

        ...
}
```

Function func_1 is used 3 times

Based on a threshold of 66% these functions would be flagged not to use the return type

FOR LOOP STEP ALIGNMENT

Memory access validation is required before a memory copy, read or write is performed to prevent attackers to get unintended access. For an accurate validation, validation addresses need to be aligned with the full read or write memory range. In example below, the validation at orange arrow needs to be performed to validate copy, read or write into Page 3.



Code example

```
void for_func(int start, int length, int step) {

int a;
        for (a = start; a < end; a += step) {
            //finding: step size not aligned
            check_func(a);
        }
        for (a = start; a < end; a += step) {
            //no finding: step used in loop
            func(a, step);
        }
        for (a = start; a < end; a += step) {
            //no finding: step size aligned
            check_func(a);
            step = MIN(end - a - 1, step);
        }
}
```

EMPTY BRANCH

Empty branches need to be avoided. Attackers can make use of empty branches by jumping to this part of the code and in that way not triggering any of the intended behavior in one of the other branches

- ✓ This check searches for empty branches in the code base
- ✓ The check can be configured to omit empty branches with a comment or a TODO marker

Code example

```
int* func(int* p, int* k) {
        if (!p) {
                NULL; // Empty branch, unused value
        } else if (!k) {
                return NULL; +
        } else {
                //ignore
                // Comment prevents warning
        }
}
```

UNEXPECTED RETURN VALUE

The return value of a function may indicate a function success or failure. For each function exit, this return value needs to represent the correct status. An update of this return value may be missing by accident. This can cause the caller of a function to make the wrong decision and give an attacker an opportunity to get unintended access

- ✓ This check can operate on complex structures within a function to determine missing or unintended return values
- ✓ Conditional branches are taken into account

Code example

```
bool func(int* p) {
   bool res;
   int* c;

   res = f1(p);          // define return value
   if (res != true) {    // conditional branch depends on
                         // return value
      return res;        // returns without return value update
                         // (correct res != SUCCESS)
   }
   C = f2(p);
   if (!c) {
      return res;        // returns without return value update
                         // (incorrect res == SUCCESS)
   }
   res = f3(p);
   return res;           // returns without return value update
}
```

LOW HAMMING DISTANCE RETURN VALUES

Fault injection is a technique to influence code execution by physically glitching the machine during program execution. Examples of physical glitches are voltage glitching, clock glitching, electromagnetic or laser pulses. A potential fault is bit flipping (one to zero or vice versa) of bus or register data. Controlled flipping of multiple bits becomes more difficult if the number of bits increases.

- ✓ Prevent fault injection attacks to influence important decisions in your software
- ✓ Set the minimal threshold on hamming distance to find exploitable vulnerabilities

Code example

```
bool func(int k, int l) {

if (k > m) {
        return false;  //return value set to false (0x0)
        }
        else {
        return true;  //return value set to true (0x1)
        }
}
```

Vulnerability will be found when minimal hamming distance is set to 2

DOUBLE CHECK FUNCTION RETURN STATUS

Fault injection is a technique to influence code execution by physically glitching the machine during program execution. A potential fault is instruction skipping like skipping a conditional branch instruction For skipping a specific instruction, precise glitch timing is essential. Obviously, skipping two specific conditional branch instructions is more difficult than one.

- ✓ Prevent fault injection attacks to skip important decisions in your software
- ✓ Check for single variable checks leading to a function exit
- ✓ Improve the robustness of your code against advanced attacks

Code example

```
bool func(int k, int l) {
    if (k > 0) {           //single check of variable 'k'
         return true;      //Exit with return value 'true'
    } else {
         return false;     //ignoring; no sensitive constant
    }
    if (l == 0) {
        if (l != 0) {      //double check of variable 'l'
            return false; //ignoring; no sensitive constant
        } else {
            return true;  //ignoring; double checked
        }
    }
    return false;          //ignoring; no sensitive constant
}
```

! See all found vulnerabilities integrated in your development environment.
True Code is integrated with Eclipse IDE

Riscure B.V.
Frontier Building, Delftechpark 49
2628 XJ Delft
The Netherlands
Phone: +31 15 251 40 90
www.riscure.com

Riscure North America
550 Kearny St., Suite 330
San Francisco, CA 94108 USA
Phone: +1 650 646 99 79
inforequest@riscure.com

Riscure China
Room 2030-31, No. 989, Changle Road, Shanghai 200031
China
Phone: +86 21 5117 5435
inforcn@riscure.com

riscure

Challenge your security