# 시큐어코딩

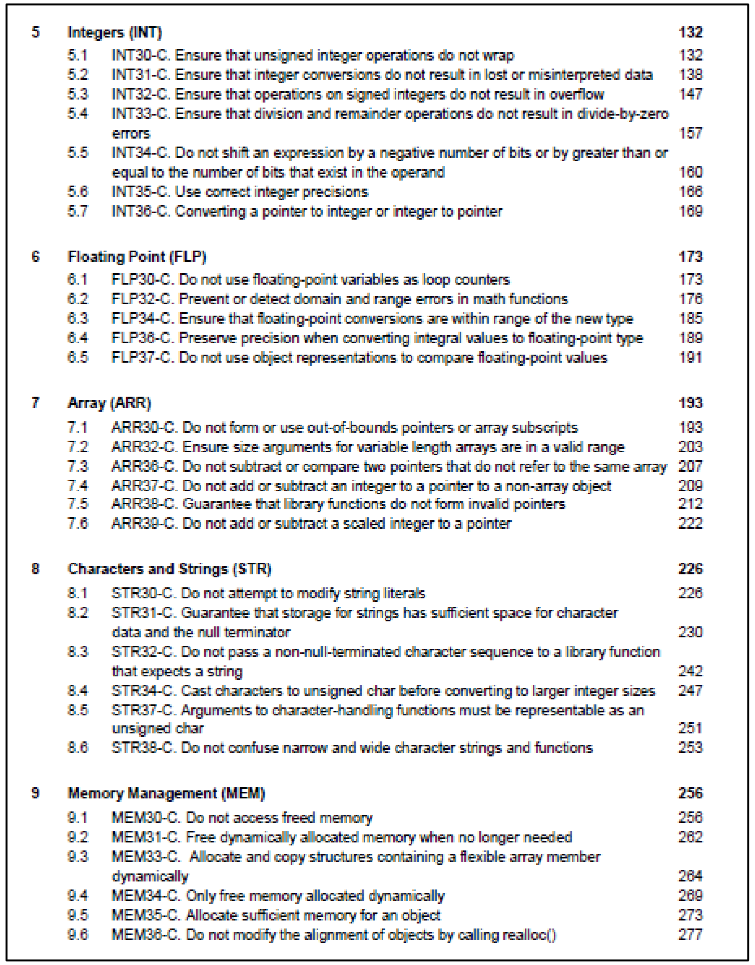
2019년 9월 30일

# 취약점 (Vulnerability) vs. 보안약점 (Weakness

- 보안약점 (weakness)
  - 소프트웨어 내의 bug, error, mistake, flaw, 등등을 총칭

- 취약점 (Vulnerability)
  - 각 기관마다 정의가 약간씩 다름 (그렇지만, 의미는 동일함)
  - 다음의 3가지 속성을 만족함
    1. 보안약점
    2. 공격자가 접근 가능하여야 함
    3. 시스템 보증 (System Assurance) 를 깸 (exploit)

# CERT Coding Standard for C
## Developing Safe, Reliable and Secure Software

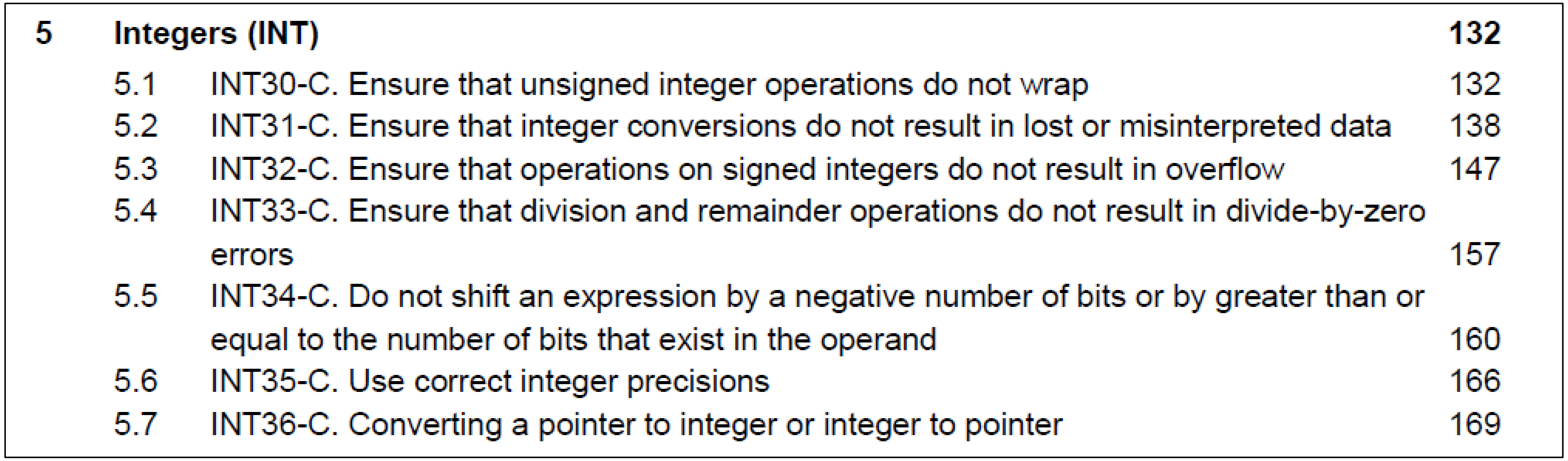


8개의 부분으로 구성

1. Preprocessor (PRE)
2. Declaration (DCL)
3. Expressions (EXP)
4. Integers (INT)
5. Floating Points (FLP)
6. Arrays (ARR)
7. Characters and Strings (STR)
8. Memory Management (MEM)

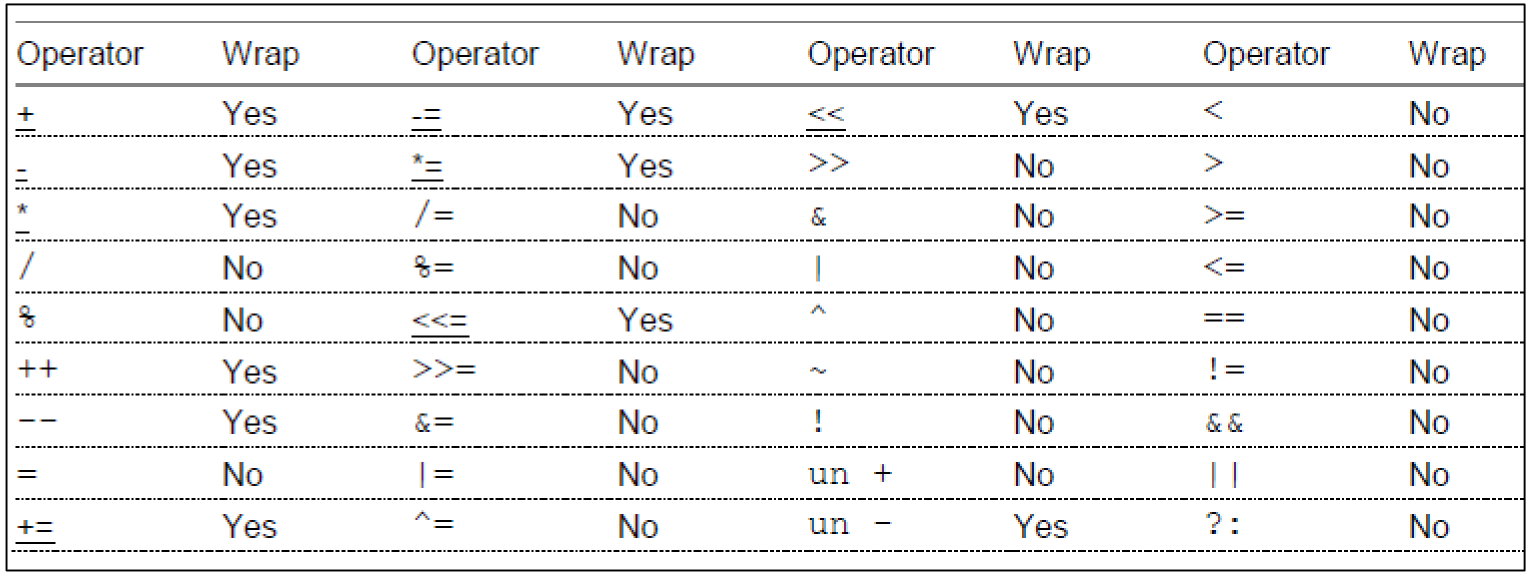
# CERT Coding Standard for C
## Developing Safe, Reliable and Secure Software

- Integers (INT)



# INT30-C

Ensure that unsigned integer operation do not wrap



| Operator | Wrap | Operator | Wrap | Operator | Wrap | Operator | Wrap |
|---|---|---|---|---|---|---|---|
| + | Yes | -= | Yes | << | Yes | < | No |
| - | Yes | *= | Yes | >> | No | > | No |
| * | Yes | /= | No | & | No | >= | No |
| / | No | %= | No | \| | No | <= | No |
| % | No | <<= | Yes | ^ | No | == | No |
| ++ | Yes | >>= | No | ~ | No | != | No |
| -- | Yes | &= | No | ! | No | && | No |
| = | No | \|= | No | un + | No | \|\| | No |
| += | Yes | ^= | No | un - | Yes | ?: | No |

- C Standard 6.2.5 (ISO/IEC 9899:2011)
- A computation involving unsigned operands can never overflow, because a result that cannot be represented by the resulting unsigned integer types is reduced modulo the number that is one greater than the largest value that can be represented by the result

# INT30-C

Ensure that unsigned integer operation do not wrap

비준수 코드 (덧셈)

```
void func(unsigned int ui_a, unsigned int ui_b) {
  unsigned int usum = ui_a + ui_b;
  /* ... */
}
```

준수 코드

```
#include <limits.h>

void func(unsigned int ui_a, unsigned int ui_b) {
  unsigned int usum;
  if (UINT_MAX - ui_a < ui_b) {
    /* Handle error */
  } else {
    usum = ui_a + ui_b;
  }
  /* ... */
}
```

```
void func(unsigned int ui_a, unsigned int ui_b) {
  unsigned int usum = ui_a + ui_b;
  if (usum < ui_a) {
    /* Handle error */
  }
  /* ... */
}
```

# INT30-C

Ensure that unsigned integer operation do not wrap

비준수 코드 (뺄셈)

```
void func(unsigned int ui_a, unsigned int ui_b) {
  unsigned int udiff = ui_a - ui_b;
  /* ... */
}
```

준수 코드

```
void func(unsigned int ui_a, unsigned int ui_b) {
  unsigned int udiff;
  if (ui_a < ui_b){
    /* Handle error */
  } else {
    udiff = ui_a - ui_b;
  }
  /* ... */
}
```

```
void func(unsigned int ui_a, unsigned int ui_b) {
  unsigned int udiff = ui_a - ui_b;
  if (udiff > ui_a) {
    /* Handle error */
  }
  /* ... */
}
```

# INT30-C

Ensure that unsigned integer operation do not wrap

비준수 코드 (곱셈)

```
pen->num_vertices = _cairo_pen_vertices_needed(
  gstate->tolerance, radius, &gstate->ctm
);
pen->vertices = malloc(
  pen->num_vertices * sizeof(cairo_pen_vertex_t)
);
```

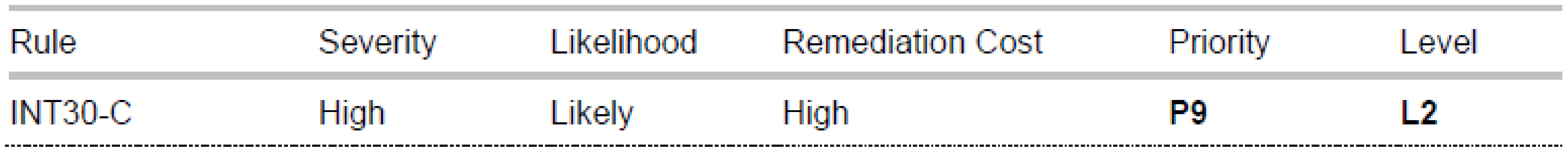
준수 코드

```
pen->num_vertices = _cairo_pen_vertices_needed(
  gstate->tolerance, radius, &gstate->ctm
);

if (pen->num vertices > SIZE MAX / sizeof(cairo pen vertex t)) {
  /* Handle error */
}
pen->vertices = malloc(
  pen->num_vertices * sizeof(cairo_pen_vertex_t)
);
```

# INT30-C

Ensure that unsigned integer operation do not wrap

위험 평가



| Rule | Severity | Likelihood | Remediation Cost | Priority | Level |
|---|---|---|---|---|---|
| INT30-C | High | Likely | High | **P9** | **L2** |

## INT31-C
Ensure that integer conversions do not result in lost or misinterpreted data

- Integer conversions, both implicit and explicit (using a cast), must be guaranteed not to result in lost or misinterpreted data.
  - Integer operands of any pointer arithmetic, including array indexing
  - The assignment expression for the declaration of a variable length arrary
  - etc(size_t, rsize_t).
- unsigned char 로 변환이 되는 다음의 함수에 변수를 넘길때
  - memset(), memset_s(), fprintf() 와 관련된 함수, fputc(), ungetc(), memchr()
- char 로 변환이 되는 함수
  - strchr(), strrchr()

# INT31-C

Ensure that integer conversions do not result in lost or misinterpreted data

비준수 코드

```
#include <limits.h>

void func(void) {
  unsigned long int u_a = ULONG_MAX;
  signed char sc;
  sc = (signed char)u_a; /* Cast eliminates warning */
  /* ... */
}
```

준수 코드

```
#include <limits.h>

void func(void) {
  unsigned long int u_a = ULONG_MAX;
  signed char sc;
  if (u_a <= SCHAR_MAX) {
    sc = (signed char)u_a;  /* Cast eliminates warning */
  } else {
    /* Handle error */
  }
}
```

## INT31-C

Ensure that integer conversions do not result in lost or misinterpreted data

비준수 코드

```c
#include <limits.h>

void func(void) {
  signed long int s_a = LONG_MAX;
  signed char sc = (signed char)s_a; /* Cast eliminates warning */
  /* ... */
}
```

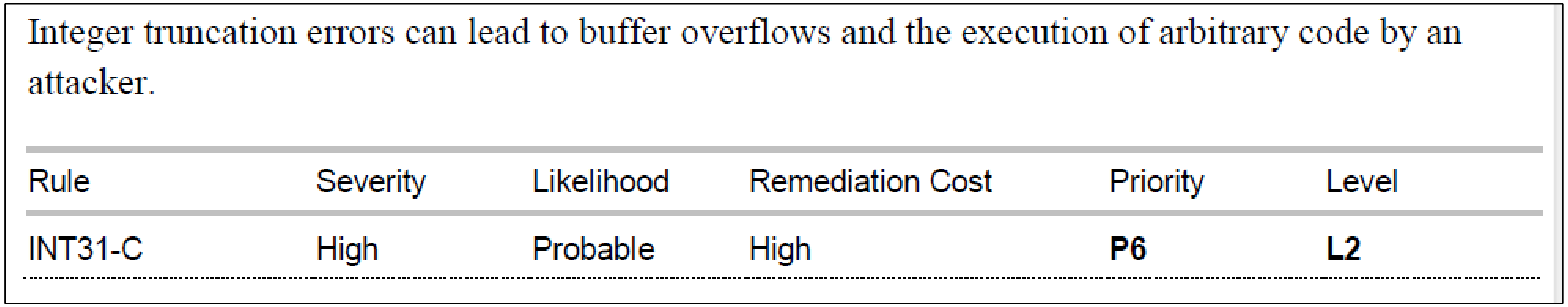
준수 코드

```c
#include <limits.h>

void func(void) {
  signed long int s_a = LONG_MAX;
  signed char sc;
  if ((s_a < SCHAR_MIN) || (s_a > SCHAR_MAX)) {
    /* Handle error */
  } else {
    sc = (signed char)s_a; /* Use cast to eliminate warning */
  }
  /* ... */
}
```

# INT31-C

Ensure that integer conversions do not result in lost or misinterpreted data

위험 평가



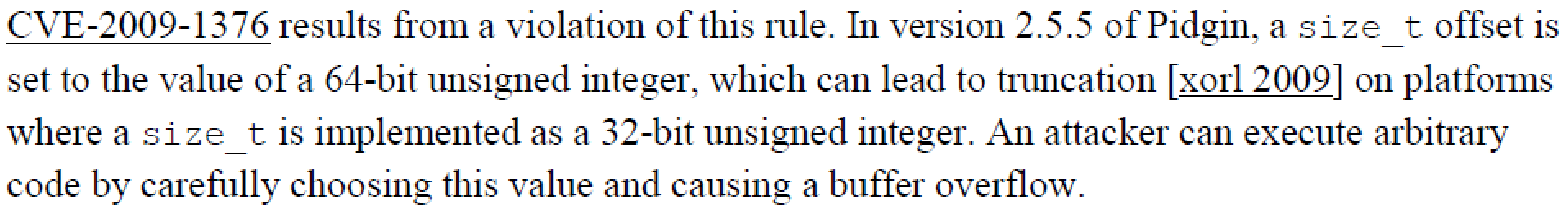
Integer truncation errors can lead to buffer overflows and the execution of arbitrary code by an attacker.

| Rule | Severity | Likelihood | Remediation Cost | Priority | Level |
|---|---|---|---|---|---|
| INT31-C | High | Probable | High | **P6** | **L2** |

# INT31-C

Ensure that integer conversions do not result in lost or misinterpreted data

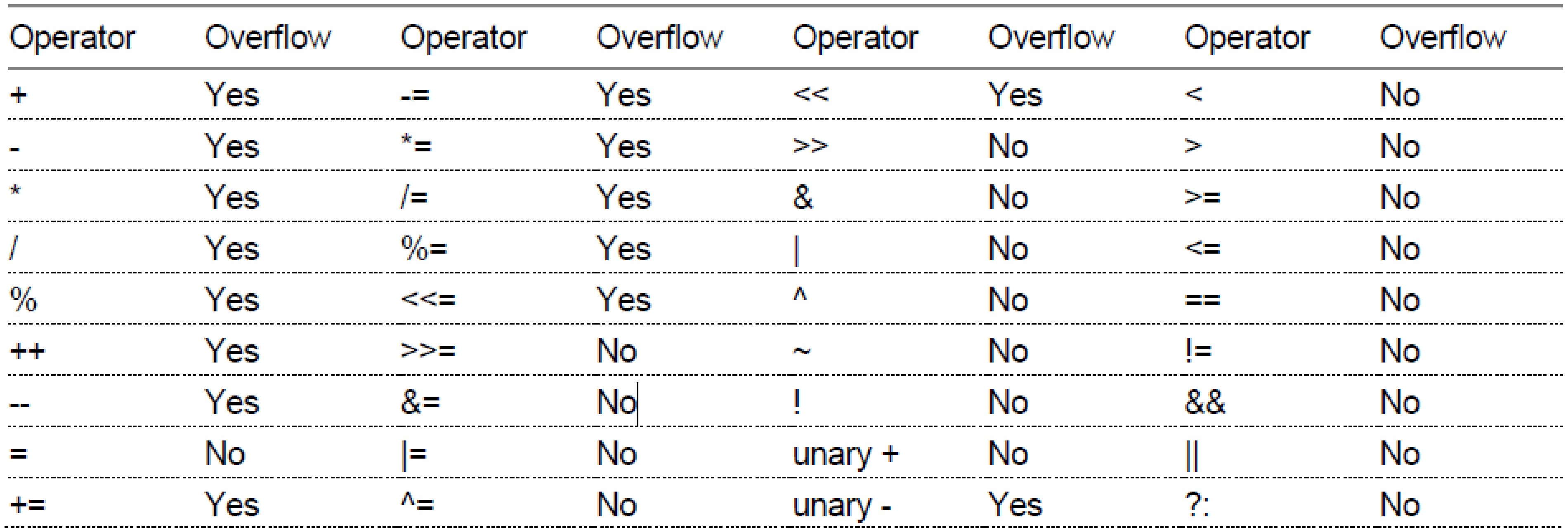
관련된 취약점



CVE-2009-1376 results from a violation of this rule. In version 2.5.5 of Pidgin, a `size_t` offset is set to the value of a 64-bit unsigned integer, which can lead to truncation [xorl 2009] on platforms where a `size_t` is implemented as a 32-bit unsigned integer. An attacker can execute arbitrary code by carefully choosing this value and causing a buffer overflow.

# INT32-C

Ensure that operations on signed integers do not result in overflow



| Operator | Overflow | Operator | Overflow | Operator | Overflow | Operator | Overflow |
|---|---|---|---|---|---|---|---|
| + | Yes | -= | Yes | << | Yes | < | No |
| - | Yes | *= | Yes | >> | No | > | No |
| * | Yes | /= | Yes | & | No | >= | No |
| / | Yes | %= | Yes | \| | No | <= | No |
| % | Yes | <<= | Yes | ^ | No | == | No |
| ++ | Yes | >>= | No | ~ | No | != | No |
| -- | Yes | &= | No | ! | No | && | No |
| = | No | \|= | No | unary + | No | \|\| | No |
| += | Yes | ^= | No | unary - | Yes | ?: | No |

- Signed integer overflow is undefined behavior 36.
- The C standard defines the behavior of arithmetic on atomic signed integer types to use two's complement representation with silent wraparound on overflow.

# INT32-C

Ensure that operations on signed integers do not result in overflow

비준수 코드

```
void func(signed int si_a, signed int si_b) {
  signed int sum = si_a + si_b;
  /* ... */
}
```

준수 코드

```
#include <limits.h>

void f(signed int si_a, signed int si_b) {
  signed int sum;
  if (((si_b > 0) && (si_a > (INT_MAX - si_b))) ||
      ((si_b < 0) && (si_a < (INT_MIN - si_b)))) {
    /* Handle error */
  } else {
    sum = si_a + si_b;
  }
  /* ... */
}
```

## INT32-C

Ensure that operations on signed integers do not result in overflow

비준수 코드

```
void func(signed int si_a, signed int si_b) {
  signed int diff = si_a - si_b;
  /* ... */
}
```

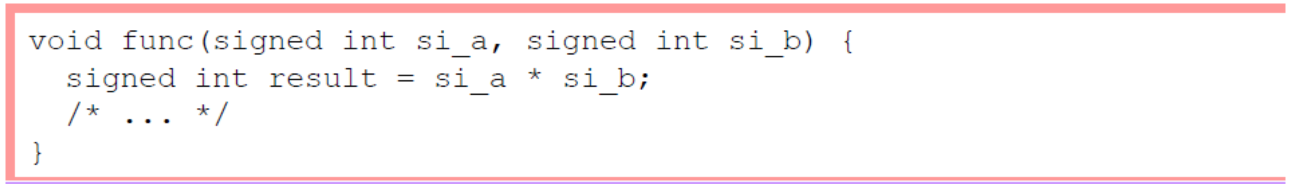
준수 코드

```
#include <limits.h>

void func(signed int si_a, signed int si_b) {
  signed int diff;
  if ((si_b > 0 && si_a < INT_MIN + si_b) ||
      (si_b < 0 && si_a > INT_MAX + si_b)) {
    /* Handle error */
  } else {
    diff = si_a - si_b;
  }

  /* ... */
}
```

# INT32-C

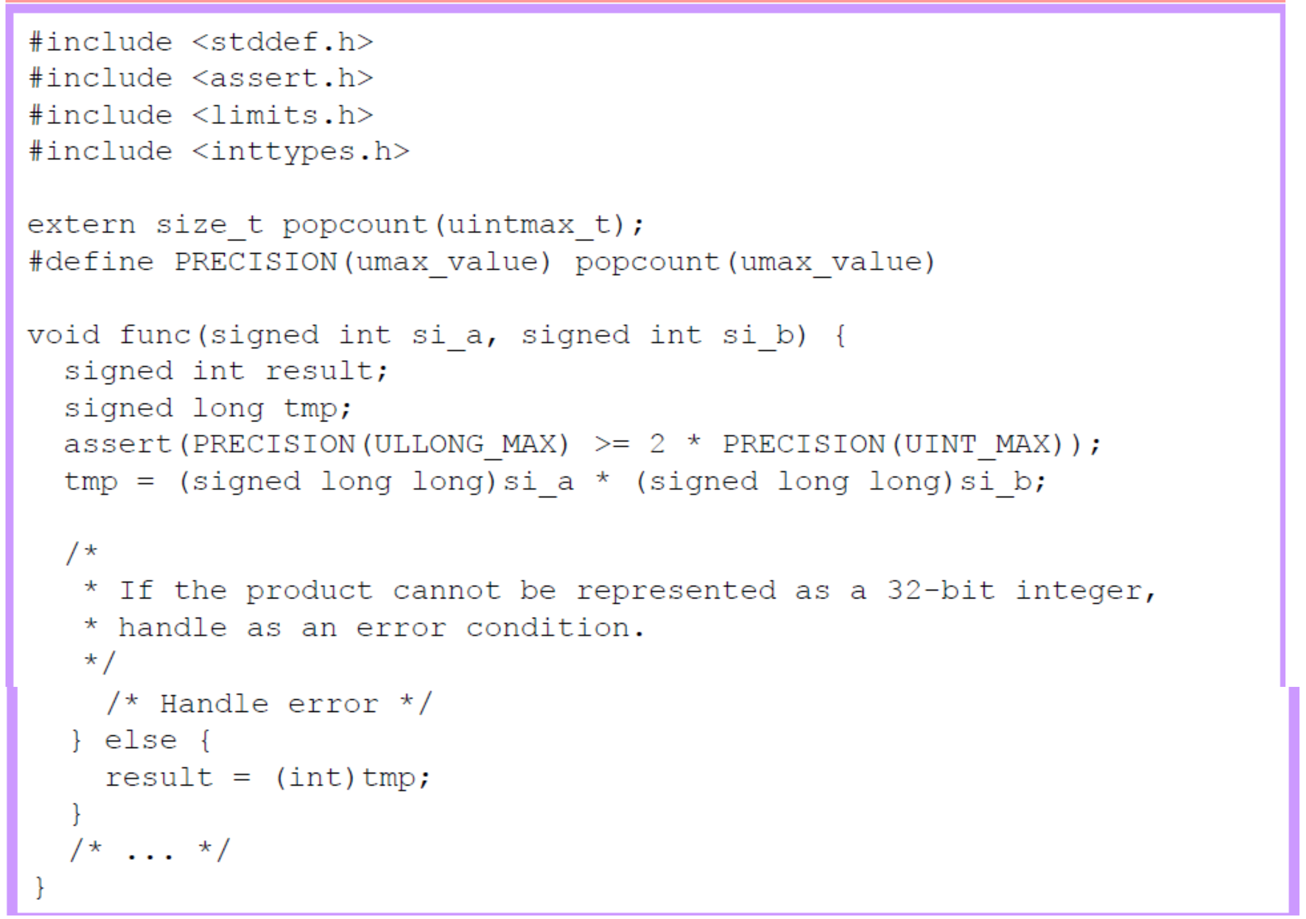
Ensure that operations on signed integers do not result in overflow

비준수 코드



```
void func(signed int si_a, signed int si_b) {
  signed int result = si_a * si_b;
  /* ... */
}
```

준수 코드



```
#include <stddef.h>
#include <assert.h>
#include <limits.h>
#include <inttypes.h>

extern size_t popcount(uintmax_t);
#define PRECISION(umax_value) popcount(umax_value)

void func(signed int si_a, signed int si_b) {
  signed int result;
  signed long tmp;
  assert(PRECISION(ULLONG_MAX) >= 2 * PRECISION(UINT_MAX));
  tmp = (signed long long)si_a * (signed long long)si_b;

  /*
   * If the product cannot be represented as a 32-bit integer,
   * handle as an error condition.
   */
    /* Handle error */
  } else {
    result = (int)tmp;
  }
  /* ... */
}
```

# INT32-C

Ensure that operations on signed integers do not result in overflow

비준수 코드 (또는)

```
void func(signed int si_a, signed int si_b) {
  signed int result = si_a * si_b;
  /* ... */
}
```

준수 코드

```
#include <limits.h>

void func(signed int si_a, signed int si_b) {
  signed int result;
  if (si_a > 0) {  /* si_a is positive */
    if (si_b > 0) {  /* si_a and si_b are positive */
      if (si_a > (INT_MAX / si_b)) {
        /* Handle error */
      }
    } else { /* si_a positive, si_b nonpositive */
      if (si_b < (INT_MIN / si_a)) {
        /* Handle error */
      }
    } /* si_a positive, si_b nonpositive */
  } else { /* si_a is nonpositive */
    if (si_b > 0) { /* si_a is nonpositive, si_b is positive */
      if (si_a < (INT_MIN / si_b)) {
        /* Handle error */
      }
    } else { /* si_a and si_b are nonpositive */
      if ( (si_a != 0) && (si_b < (INT_MAX / si_a))) {
        /* Handle error */
      }
    } /* End if si_a and si_b are nonpositive */
  } /* End if si_a is nonpositive */

  result = si_a * si_b;
}
```

## INT32-C

Ensure that operations on signed integers do not result in overflow

비준수 코드

```
void func(signed long s_a, signed long s_b) {
  signed long result;
  if (s_b == 0) {
    /* Handle error */
  } else {
    result = s_a / s_b;
  }
  /* ... */
}
```

준수 코드

```
#include <limits.h>

void func(signed long s_a, signed long s_b) {
  signed long result;
  if ((s_b == 0) || ((s_a == LONG_MIN) && (s_b == -1))) {
    /* Handle error */
  } else {
    result = s_a / s_b;
  }
  /* ... */
}
```

# INT32-C

Ensure that operations on signed integers do not result in overflow

비준수 코드

```
void func(signed long s_a, signed long s_b) {
  signed long result;
  if (s_b == 0) {
    /* Handle error */
  } else {
    result = s_a % s_b;
  }
  /* ... */
}
```

준수 코드

```
#include <limits.h>

void func(signed long s_a, signed long s_b) {
  signed long result;
  if ((s_b == 0 ) || ((s_a == LONG_MIN) && (s_b == -1))) {
    /* Handle error */
  } else {
    result = s_a % s_b;
  }
  /* ... */
}
```

# INT32-C

Ensure that operations on signed integers do not result in overflow

비준수 코드

```
void func(signed long s_a) {
  signed long result = -s_a;
  /* ... */
}
```

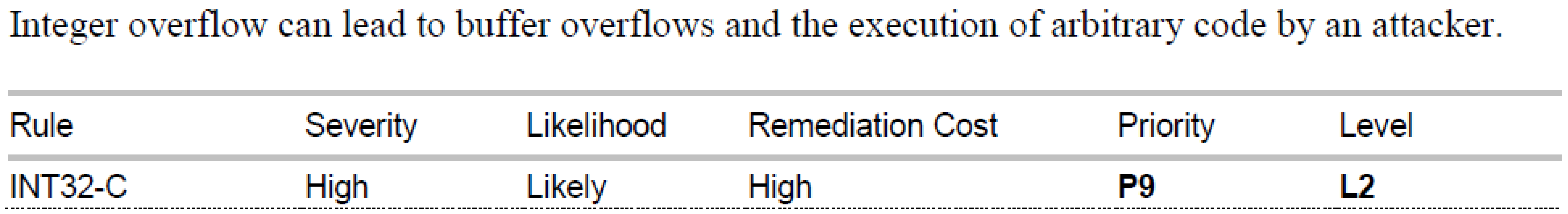
준수 코드

```
#include <limits.h>

void func(signed long s_a) {
  signed long result;
  if (s_a == LONG_MIN) {
    /* Handle error */
  } else {
    result = -s_a;
  }
  /* ... */
}
```

# INT32-C

## Ensure that operations on signed integers do not result in overflow

Risk Assessment



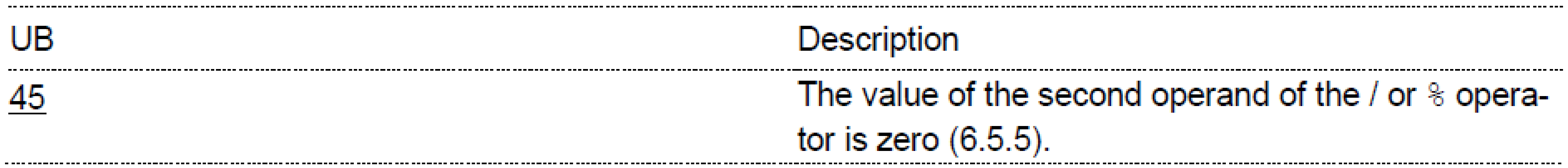
Integer overflow can lead to buffer overflows and the execution of arbitrary code by an attacker.

| Rule | Severity | Likelihood | Remediation Cost | Priority | Level |
|---|---|---|---|---|---|
| INT32-C | High | Likely | High | **P9** | **L2** |

# INT33-C

Ensure that division and remainder operations do not result in devide-by-zero errors

- The C standard identifies the following condition under which devision and remainder operations result in undefined behavior (UB)



| UB | Description |
|---|---|
| 45 | The value of the second operand of the / or % opera-tor is zero (6.5.5). |

## INT33-C

Ensure that division and remainder operations do not result in devide-by-zero errors

비준수 코드

```
#include <limits.h>

void func(signed long s_a, signed long s_b) {
  signed long result;
  if ((s_a == LONG_MIN) && (s_b == -1)) {
    /* Handle error */
  } else {
    result = s_a / s_b;
  }
  /* ... */
}
```

준수 코드

```
#include <limits.h>

void func(signed long s_a, signed long s_b) {
  signed long result;
  if ((s_b == 0) || ((s_a == LONG_MIN) && (s_b == -1))) {
    /* Handle error */
  } else {
    result = s_a / s_b;
  }
  /* ... */
}
```

# INT33-C

Ensure that division and remainder operations do not result in devide-by-zero errors

비준수 코드

```
#include <limits.h>

void func(signed long s_a, signed long s_b) {
  signed long result;
  if ((s_a == LONG_MIN) && (s_b == -1)) {
    /* Handle error */
  } else {
    result = s_a % s_b;
  }
  /* ... */
}
```

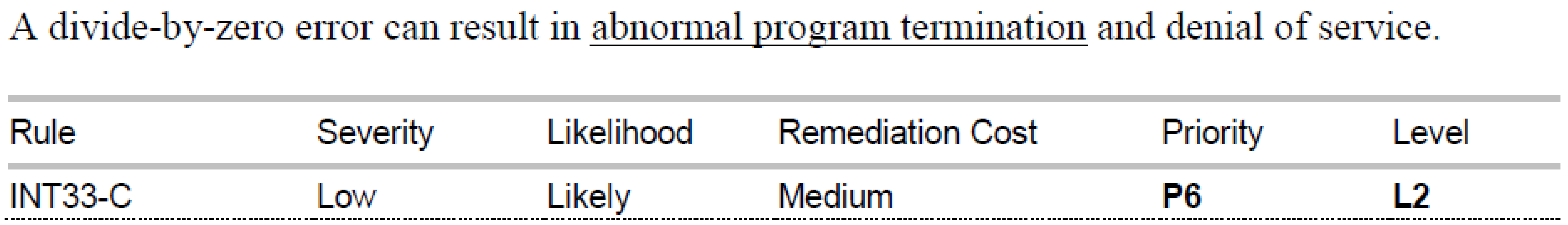
준수 코드

```
#include <limits.h>

void func(signed long s_a, signed long s_b) {
  signed long result;
  if ((s_b == 0 ) || ((s_a == LONG_MIN) && (s_b == -1))) {
    /* Handle error */
  } else {
    result = s_a % s_b;
  }
  /* ... */
}
```

# INT33-C

Ensure that division and remainder operations do not result in devide-by-zero errors

Risk Assessment



A divide-by-zero error can result in abnormal program termination and denial of service.

| Rule | Severity | Likelihood | Remediation Cost | Priority | Level |
|---|---|---|---|---|---|
| INT33-C | Low | Likely | Medium | **P6** | **L2** |

## 다음 주

- 다음 주는 Polyspace 도구 강의가 있습니다. 학교 ftp 사이트에서 download 받고, install 하여 가급적 notebook 을 지참하고 수업에 참가해 주시면 감사하겠습니다.

## 해보기

- INT 부분에서 비준수 코드를 작성하고 결과가 이상하게 나오는 프로그램 예를 만들어 실행하고 그 결과 프린트하기. 그리고 동일한 입력을 이용해서 준수코드로 변환해서 실행하고 그 결과를 출력하기
- 단, INT30-C, INT31-C, ~ INT336-C 에서 예제를 두개씩 선택해서 프로그램 하기.
- 그리고 polyspace 로 정적분석 해 봅시다.