**بسمه تعالی**

# عنوان مستند:

## برنامه نویسی امن C++

مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

**مرکز ماهر**

**تابستان ۹۵**

فهرست مطالب


**۱ مقدمه ..... ۱**

۱-۱ امنیت نرم‌افزار: چالش اصلی ..... ۱

۱-۲ رشد آسیب‌پذیری نرم‌افزارها ..... ۲

۱-۳ راه‌حل ..... ۳

۱-۴ رفتار تعریف‌نشده ..... ۴

**۲ مباحث کلی ..... ۵**

۲-۱ توابع بدون نشانوند ..... ۵

۲-۲ اعداد تصادفی ..... ۵

۲-۳ عدم تعریف و مقداردهی اولیه متغیر قبل از اولین case در switch ..... ۶

۲-۴ عدم استفاده از توابع بازگشتی ورودی به تابع ..... ۸

۲-۵ الحاق مضاعف ..... ۹

۲-۶ اصول رمزنگاری ..... ۱۱

۲-۶-۱ رمزنگاری یک‌طرفه ..... ۱۲

۲-۶-۲ رمزنگاری با کلید متقارن ..... ۱۲

۲-۶-۳ رمزنگاری با کلید عمومی (نامتقارن) ..... ۱۳

۲-۷ کدگذاری دودویی به متن ..... ۱۴

**۳ بررسی ایمن داده‌ها و اشیاء ..... ۱۵**

۳-۱ مدیریت مقدار و نوع داده‌های بنیادی ..... ۱۵

۳-۲ مدیریت مقدار و نوع داده‌های بنیادی: مقداردهی اولیه ..... ۱۵

۳-۳ مدیریت مقدار و نوع داده‌های بنیادی: مقداردهی اولیه آرایه‌ها ..... ۱۶

**۴ بررسی ایمن داده‌ها و اشیاء ..... ۱۸**

۴-۱ قالب: تغییر قالب پایدار ..... ۱۸

**۵ اشاره‌گر ..... ۲۰**

۵-۱ مقداردهی اولیه ..... ۲۰

۵-۲ اشاره‌گر تهی ..... ۲۱

۵-۳ اشاره‌گر معلق ..... ۲۳

۵-۴ اشاره‌گر رام نشده ..... ۲۵

۵-۵ اشاره‌گر هوشمند ..... ۲۶

**۶ حافظه ..... ۲۹**

۶-۱ نشت حافظه ..... ۲۹

۶-۲ آزادسازی مضاعف ..... ۳۲

۶-۳ آزادسازی اشاره‌گر تهی ..... ۳۴


مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه‌ای

۶-۴ پاک‌سازی ایمن، صفر سازی حافظه ..... ۳۵
۶-۵ فاش شدن اطلاعات ..... ۳۶
۶-۶ ویرایش حافظه ..... ۴۰
۶-۷ سرریز بافر ..... ۴۲
۶-۸ قالب رشته بازرسی نشده ..... ۴۵
۶-۹ سرریز عدد صحیح ..... ۴۷
۶-۱۰ خرابی حافظه ..... ۴۹
**7 هم‌زمانی ..... ۵۰**
۷-۱ ایمنی نخ‌ها ..... ۵۰
۷-۲ وضعیت پیشی گرفتن ..... ۵۰
۷-۳ بن‌بست ..... ۵۵
**۸ مدیریت خطا ..... ۵۸**
۸-۱ خطاهای منطقی ..... ۵۸
۸-۲ خطاهای زمان اجرا ..... ۵۹
**۹ مدیریت ارتباطات برون برنامه‌ای: ..... ۶۴**
۹-۱ ارتباط با پایگاه داده ..... ۶۴
۹-۲ اجرای برنامه‌های بیرونی ..... ۶۷
۹-۳ پرونده‌های موقت ..... ۷۱
۹-۴ زمان بازرسی، زمان استفاده ..... ۷۴
۹-۵ رمزنگاری داده‌های ورودی و خروجی ..... ۷۵
۹-۶ ایمنی ارتباطات شبکه‌ای ..... ۷۷
۹-۷ انتقال اطلاعات بین سیستمی ..... ۷۹
**10 نکات برنامه‌نویسی ایمن و تدافعی ..... ۸۰**
۱۰-۱ نکات نگارشی و دستوری ..... ۸۰
۱۰-۲ نکات مختص سیستم‌عامل خانواده Unix ..... ۸۸
**۱۱ پیوست ..... ۹۰**
۱۱-۱ خانواده printf ..... ۹۰
۱۱-۲ atomic ..... ۹۲
۱۱-۳ ثابت‌های محافظ حافظه ..... ۹۳
۱۱-۴ توابع تأثیرگذار بر errno ..... ۹۵
۱۱-۵ بازه و دامنه توابع ریاضی ..... ۹۶
۱۱-۶ خطاهای رایج توابع موجود در زبان C/C++ ..... ۹۷
**۱۲ ابزارهای مورداستفاده ..... ۱۰۴**

مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه‌ای

۱۳ منابع.....۱۰۵

مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

# ۱ مقدمه

## ۱-۱ امنیت نرم‌افزار: چالش اصلی

با گسترش روزافزون نرم‌افزارها بر روی پلتفرم‌های مختلف، امنیت نرم‌افزار هرروزه از اهمیت بیشتری برخوردار می‌شود.

وجود آسیب‌پذیری در یک نرم‌افزار می‌تواند اهداف یک حمله‌کننده را به نتیجه برساند، حال این هدف می‌تواند دسترسی به اطلاعات شخصی یک فرد در رایانه، دسترسی به شبکه خصوصی یک شرکت، سرقت اطلاعات موجود در تلفن همراه، مختل سازی یک سرور و صدها هدف دیگر باشد.

شاید یک برنامه به‌ظاهر کم‌اهمیت و ساده جلوه داده شود اما وجود آسیب‌پذیری در آن ممکن است به خطری جدی برای استفاده‌کنندگان از آن تبدیل شود چراکه همان آسیب‌پذیری برای برآورده کردن اهداف یک حمله‌کننده کافی است. لذا برنامه‌نویسی ایمن یک برنامه در هر سطح و ابعادی که باشد، امری ضروری است.

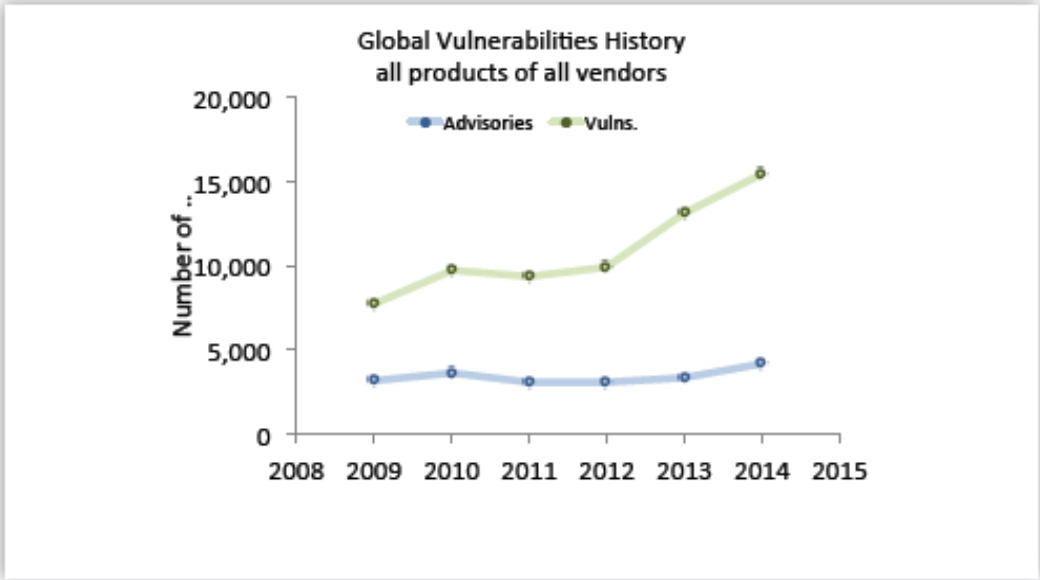
با توجه به اهمیت امنیت نرم‌افزار، شرکت‌های بزرگ دنیا به ارائه راهکارهایی چون طراحی زبان‌ها و محیط‌های برنامه‌نویسی و مفسر و مترجم‌هایی باقابلیت‌های کنترل امنیتی و همچنین راهکارهای امنیتی بر روی سیستم‌عامل و بسیاری از راهکارهای دیگر پرداخته‌اند اما با توجه به عدم توانایی راهکارها در کنترل تمامی موارد امنیتی، عدم امکان پیاده‌سازی راهکارهای امنیتی بر روی برخی ساختارها، ایجاد محدودیت برای دسترسی به برخی منابع و امکانات و مشکلات کوچک و بزرگ دیگر، برنامه‌نویسی یک برنامه به‌صورت ایمن بهترین راهکار برای محافظت از یک برنامه است.

یکی از زبان‌هایی که در کنار محبوبیت در میان برنامه‌نویسان، همیشه یکی از زبان‌های پرچالش در برنامه‌نویسی ایمن بوده است، خانواده زبان‌های C به‌خصوص C++ است.

در این زبان‌ها عمده مدیریت منابع به برنامه‌نویس واگذارشده که در صورت عدم مدیریت درست آن‌ها، آسیب‌پذیری‌های مختلفی رخ می‌دهد.

## ۲-۱ رشد آسیب‌پذیری نرم‌افزارها

بر طبق گزارش‌ها، آسیب‌پذیری‌های گزارش‌شده در ۵ سال اخیر ۵۵٪ رشد داشته‌اند. همچنین آسیب‌پذیری‌های گزارش‌شده با رشد ۱۱٪ ای در سال ۲۰۱۴ به نسبت سال ۲۰۱۳ همراه بوده‌اند.




این آمار نشان‌دهنده روند صعودی تعداد آسیب‌پذیری‌های گزارش‌شده در سال‌های اخیر است که البته رشد کمی نرم‌افزارها در این بازه نیز قابل‌توجه است.

بدیهی است که این روند افزایشی زنگ خطری برای استفاده‌کنندگان از نرم‌افزارها است.

برنامه‌های نوشته‌شده با زبان برنامه‌نویسی C/C++ در این آمار قابل‌توجه‌اند چراکه این زبان‌ها جدا از محبوبیت، در اکثر محیط‌های برنامه‌نویسی و مفسرها کنترل کامل روند اجرایی یک برنامه و تخصیص منابع را به برنامه‌نویس سپرده که در صورت عدم مدیریت صحیح این موارد، برنامه با مخاطرات امنیتی جدی روبروست.

---

https://secunia.com/?action=fetch&filename=secunia_vulnerability_review_2015_pdf.pdf[1]

در این میان، نیاز است که برنامه‌نویسان این زبان‌ها با این مخاطرات آشنا شده و برنامه را بر مبنای اصول امنیتی طراحی کنند.

## ۳-۱ راه‌حل

همان‌طور که اشاره شد، راهکارهای بسیار زیادی برای ایجاد امنیت نرم‌افزار ارائه‌شده‌اند، همچنین شرکت‌های بزرگ برای آزمودن و برقراری امنیت نرم‌افزار خود بخشی خاص ایجاد کرده‌اند و یا نرم‌افزار را برای آزمودن به شرکت‌هایی که این خدمات را انجام می‌دهند ارسال می‌کنند. همچنین برنامه‌ها را توسط ابزارهایی خاص مورد آزمودن امنیتی قرار می‌دهند تا در صورت وجود آسیب‌پذیری از آن مطلع گردند اما هیچ‌کدام از راهکارهای ذکرشده به‌طور کامل کارآمد نیست.

بهترین راهکار برای جلوگیری از بروز آسیب‌پذیری نرم‌افزارها، برنامه‌نویسی پدافندی و ایمن آن نرم‌افزار از ابتداست.

رویکرد فعلی در برقراری امنیت کامل یک نرم‌افزار امروزه برنامه‌نویسی پدافندی و ایمن آن برنامه از پایه است و امروزه یکی از ارکان اصلی شرکت‌های نرم‌افزاری آموزش برنامه‌نویسی پدافندی و ایمن به برنامه‌نویسان است.

همچنین شرکت‌های ارائه‌دهنده راهکارهای امنیتی نیز سرویس‌های آموزش برنامه‌نویسی پدافندی و ایمن را به برنامه‌نویسان و شرکت‌های نرم‌افزاری ارائه می‌کنند.

برنامه‌نویسی پدافندی و ایمن یک برنامه بدین معنی نیست که فقط از بروز آسیب‌پذیری‌ها جلوگیری کرد، بلکه راهکاری برای نوشتن یک برنامه بر مبنای اصول برنامه‌نویسی در کنار توجه به فاکتورهای امنیتی است حتی اگر یک فاکتور به‌صورت مستقیم با آسیب‌پذیری مرتبط نباشد و یک ریسک امنیتی محسوب نشود.

در این مستندات، با توجه به جامعیت و کاربرد فراوان زبان C++ در کنار محبوبیت، مباحث و نکات اساسی در برنامه‌نویسی پدافندی و ایمن این زبان مطرح‌شده و انواع آسیب‌پذیری و شیوه جلوگیری از بروز آن‌ها و رفع آن‌ها در صورت بروز، توضیح داده می‌شود. همچنین سعی می‌شود تا راهکارهای ارائه‌شده تا حد امکان قابل پیاده‌سازی در زبان C نیز باشند. با توجه به گستردگی ابزارهای برنامه‌نویسی این زبان و وجود کامپایلرهای مختلف، زبان معیار برای این مستند مدنظر قرارگرفته شده است و ساختار ارائه‌شده مربوط به ابزار یا کامپایلر خاصی نیست اما بنا بر نیاز مثال‌هایی در کنار زبان معیار از ابزارهایی خاص نیز ارائه می‌گردد.

## ۱-٤ رفتار تعریف‌نشده[۲]

رفتار تعریف‌نشده عبارت است از رخ دادن اتفاقاتی در برنامه که موردنظر برنامه‌نویس نبوده و باعث می‌شود برنامه رفتاری غیر از رفتار آرمانی از خود نشان دهد.

رفتار تعریف‌نشده ممکن است شامل مختل شدن عمل کرد برنامه (کرش)، خروجی نامربوط و غلط، بروز آسیب‌پذیری‌های نرم‌افزاری و موارد دیگر شود.

زبان C++ یکی از زبان‌هایی است که به دلیل آزادی عمل بسیار برنامه‌نویس در آن و عدم وجود سازوکارهای کنترلی و پیش‌گیری درون کامپایلرهای رایج به‌صورت پیش‌فرض، همیشه در معرض خطر رفتار تعریف‌نشده است.

وجود رفتار نامتعارف در یک برنامه نه‌تنها امنیت خود آن برنامه، بلکه ممکن است امنیت سیستم‌عامل، شبکه را نیز به خطر بیندازد.

جلوگیری از بروز رفتارهای تعریف‌نشده و مقابله با آن‌ها و کنترل آن‌ها از مباحث مهم برنامه‌نویسی تدافعی و ایمن است.

رفتار نامتعارف به شکل‌های گوناگون ممکن است رخ دهد، یک سری از آن‌ها رایج بوده و موردبررسی قرار می‌گیرد اما یک سری دیگر از رفتارهای نامتعارف مربوط به اشتباهات برنامه‌نویسی برنامه‌نویس است که در هر برنامه و برای هر برنامه‌نویس به‌صورت متفاوت رخ می‌دهد، لذا با ارائه راهکارهای کلی، برنامه‌نویس باید راهکارها را تامیم داده و در برنامه خود از آن‌ها استفاده کند و در زمان برنامه‌نویسی به نکات مطرح‌شده توجه داشته باشد.

مرکز مدیریت امداد و هماهنگی عملیات (خدمات‌های رایانه ای)

---

[۲] Undefined Behavior

# ۲ مباحث کلی

## ۲-۱ توابع بدون نشانوند[۳]

برای تعریف توابع بدون نشانوند، باید درون پرانتزهای باز و بسته از کلیدواژه void استفاده کرد.

قطعه‌کد زیر نمونه‌ای از تعریف تابع بدون نشانوند است:

```
int func(void)
{
        return 1;
}
```

## ۲-۲ اعداد تصادفی[۴]

در صورت نیاز به تولید اعداد تصادفی، نباید از تابع rand() استفاده کرد چراکه اعداد تولیدشده توسط این تابع به‌صورت کامل تصادفی نیستند و احتمال بروز تکرار در آن‌ها بالاست.

در صورت نیاز به استفاده از rand() حتماً باید از srand() برای تعریف بذر[۵] آن استفاده کرد.

برای تولید اعداد تصادفی می‌توان از تابع CryptGenRandom() در سیستم‌عامل ویندوز و تابع random() در سیستم‌عامل لینوکس استفاده کرد. باید دقت داشت که قبل از اجرای random() باید از تابع srandom() برای تعریف بذر استفاده کرد.

تابع CryptGenRandom در هدر Wincrypt.h قرار داشته و ساختار آن به‌صورت زیر است:

```
BOOL WINAPI CryptGenRandom(
  _In_    HCRYPTPROV hProv,
  _In_    DWORD      dwLen,
  _Inout_ BYTE       *pbBuffer
);
```

---

[۳] Argument

[۴] Random

[۵] Seed

این تابع درواقع به تعداد معینی بایت تصادفی ایجاد می‌کند.

hProv یک محیا کننده سرویس رمزنگاری[۶] است که با استفاده از فراخوانی تابع CryptAcquireContext() به وجود می‌آید.

dwLen تعداد بایت موردنظراست.

pbBuffer بافر دریافت‌کننده بایت‌های ساخته‌شده است که حداقل باید هم‌اندازه تعداد بایت dwLen باشد.

قطعه کد زیر نمونه‌ای از ساخت عدد تصادفی با استفاده از این تابع است:

```
#include <windows.h>
#include <wincrypt.h>
#include <iostream>

int main(void)
{
	HCRYPTPROV prov;
	CryptAcquireContext(&prov, NULL, NULL, PROV_RSA_FULL, 0);
	long int li = 0;
	CryptGenRandom(prov, sizeof(li), (BYTE *)&li);
	printf("Random number: %ld\n", li);
	return 0;
}
```

خروجی قطعه کد بالا همانند زیر است:

```
Random number: -343953068
```

که عددی تصادفی در بازه long int است.

## ۳-۲ عدم تعریف و مقداردهی اولیه متغیر قبل از اولین case در switch

چنانچه در switch قبل از اولین case یک متغیر با مقداردهی اولیه تعریف شود، مقداردهی اولیه‌ی متغیر انجام نمی‌شود و باعث بروز رفتار تعریف‌نشده خواهد شد و مقدار تصادفی‌ای که درون حافظه قرار دارد را به‌عنوان مقدار برمی‌گرداند.

---

[۶] Cryptographic Service Provider (CSP)

قطعه کد زیرنمایان گر این حالت است:

```
#include <iostream>

int main(void)
{
	int sw = 0;
	std::cin >> sw;
	switch(sw)
	{
		int i = 4;
		case 0:
			i = 17;
		default:
			printf("%d",i);
	}
	return 0;
}
```

چنانچه عدد ورودی توسط کاربر برابر ۰ باشد عدد ۱۷ در خروجی چاپ می‌شود در غیر این صورت عدد تصادفی موجود در حافظه به نمایش درمی‌آید. خروجی قطعه کد بالا به‌صورت زیر است:

```
0
17
```

```
1
12358940
```

راهکار صحیح این است که درون ساختار switch به‌جز المان‌های مربوطه از دستورات دیگر استفاده نکرده و دستورات را به خارج از ساختار switch منتقل کنیم.

قطعه کد زیر نمونه اصلاح‌شده قطعه کد بالا است:

```
#include <iostream>

int main(void)
{
	int sw = 0;
	std::cin >> sw;
	int i = 4;
	switch(sw)
	{
		case 0:
			i = 17;
		default:
			printf("%d",i);
	}
	return 0;
}
```

خروجی قطعه کد بالا درازای ورود عدد غیر ۰ به‌صورت زیر است:

```
1
4
```

## ۲-٤ عدم استفاده از توابع بازگشتی ورودی به تابع

عدم استفاده از توابع بازگشتی ورودی به تابع در زمان ایجاد و مقداردهی اولیه یکی از شیءهای ایستا[۷]ی آن تابع نکته‌ای است که باید به آن توجه کرد.

برای بیان این مطلب، به قطعه کد زیر توجه کنید:

```
#include <stdexcept>

int fact(int i) noexcept(false) {
	if (i < 0) {
		throw std::domain_error("i must be >=0");
	}

	static const int cache[] = {
		fact (0), fact(1), fact(2), fact(3), fact(4), fact (5),
		fact (6), fact(7), fact(8), fact(9), fact(10), fact (11),
		fact (12), fact(13), fact(14), fact(15), fact(16)
	};
	if (i < (sizeof(cache) / sizeof(int))) {
		return cache[i];
	}
	return i > 0 ? i * fact (i - 1) : 1;
}
```

در زمان ساخته‌شدن و مقداردهی اولیه cache تابع fact مجدداً فراخوانی شده و این عمل به دلیل ایستا و پایدار[۸] بودن cache باعث بروز رفتار تعریف‌نشده و خطا می‌شود.

قطعه کد زیر نمونه اصلاح‌شده قطعه کد بالا است:

مرکز مدیریت امداد سامانه سیمای رخدادهای رایانه ای

---

Static[۷]

Constant[۸]

```
#include <stdexcept>

int fact(int i) noexcept(false) {
	if (i < 0) {
		throw std::domain_error("i must be >=0");
	}

	static int cache[17];
	if (i < (sizeof(cache) / sizeof(int))) {
		if (0 == cache[i]) {
			cache[i] = i > 0 ? i * fact(i - 1) : 1;
		}
		return cache[i];
	}

	return i > 0 ? i * fact(i - 1) : 1;
}
```

به این روش مقداردهی اولیه و ساخته‌شدن می‌گویند، در این قطعه کد ساختار cache از نوع پایدار نبوده و همچنین به‌جای تابع بودن، مبتنی بر یک آرایهٔ مقداردهی اولیه شده صفر[۹] است. سپس هر یک از اعضای آن در صورت عدم مقداردهی، به‌صورت بازگشتی محاسبه می‌شوند و درون آن قرار می‌گیرند. به این روش مقداردهی تنبل[۱۰] می‌گویند.

## ۵-۲ الحاق مضاعف[۱۱]

الحاق مضاعف زمانی رخ می‌دهد که یک هدر دو و یا چند بار به برنامه اضافه شود. این حالت معمولاً به دلیل الحاق هدرهای دیگر که شامل هدر الحاق شده می‌باشند رخ می‌دهد.

قطعه کد زیر نمونه‌ای از این حالت است:

---

[۹] Zero-Initialized

[۱۰] Lazy Initialization

[۱۱] Double Inclusion



مرکز مدیریت امداد هماهنگی عملیات رخدادهای رایانه ای

```
//a.h
struct a
{
	int member;
};

//b.h
#include "a.h"

//c.c
#include "a.h"
#include "b.h"
```

در C هدرهای a و b به الحاق درمی‌آیند درحالی‌که در b نیز هدر a به الحاق درمی‌آید و درنتیجه الحاق مضاعف رخ می‌دهد.

برای جلوگیری از این روش، جدای از دقت برنامه‌نویس در الحاق‌ها، از محافظ الحاق[۱۲] استفاده می‌شود.

قطعه کد زیر نمونه‌ای از پیاده‌سازی محافظ الحاق است:

```
//a.h
#ifdef A_H
#define A_H

struct a
{
	int member;
};

#endif
```

در این حالت، درصورتی‌که یک‌بار هدر به الحاق دربیاید، در دفعات بعدی شرط صادق نبوده و عملیات الحاق انجام نمی‌شود.

روش دیگر برای جلوگیری از الحاق مضاعف استفاده از pragma once است.

---

[۱۲] Include Guard



مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

این دستور پیش پردازنده[۱۳] هنوز به‌صورت استاندارد درون زبان C/C++ وارد نشده است اما تقریباً تمامی کامپایلرهای رایج به‌جز Solaris Studio C/C++ از آن پشتیبانی می‌کنند. شیوه اجرایی هرکدام متفاوت است اما درنهایت خروجی یکسانی برای جلوگیری از الحاق مضاعف می‌دهند.

قطعه کد زیر نمونه‌ای از پیاده‌سازی این روش است:

```
//a.h
#pragma once

struct a
{
	int member;
};
```

## ۲-۶ اصول رمزنگاری[۱۴]

بدیهی است که یکی از بخش‌های مهم هر برنامه داده‌های ورودی و خروجی آن است. حال این داده‌ها می‌توانند صرفاً نتیجه یک عمل ریاضی ساده باشند و یا مشخصات کارت اعتباری کاربر برنامه باشند. ازاین‌رو نیاز است که برای نگهداری و انتقال اطلاعات از محفوظ ماندن آن‌ها در مقابل اشخاص تأیید نشده جلوگیری کنیم و رمزنگاری ما را در این امر یاری می‌کند.

یکی از بهترین کتابخانه[۱۵]های رمزنگاری در زبان C++، کتابخانه Crypto++[۱۶] است. libcrypto از OpenSSL نیز کتابخانه دیگری برای انجام رمزنگاری است.

رمزنگاری به ۳ دسته اصلی تقسیم می‌شود:

مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

---

[۱۳] Preprocessor Directive

[۱۴] Cryptography

[۱۵] Library

[۱۶] https://www.cryptopp.com

### ۱-۶-۲ رمزنگاری چکیده ساز[۱۷]

در این نوع رمزنگاری، ورودی به‌گونه‌ای رمزنگاری می‌شود که امکان برگشت خروجی به ورودی اولیه غیرممکن است.

ویژگی غیرقابل‌بازگشت بودن رمزنگاری های چیده ساز این امکان را فراهم می‌سازد تا بتوان اطلاعاتی همانند کلمه عبور کاربر را ذخیره کرده و در زمان بررسی صحت ورود کلمه عبور، کلمه عبور ورودی چکیده شده را با مقدار ذخیره‌شده مقایسه کرد.

چکیده ها همچنین در اطمینان از سالم بودن پرونده، اطمینان از صحیح بودن پکت‌های ارسالی در شبکه، الگوریتم‌های جستجو و موارد دیگر کاربرد دارند.

چند نمونه از الگوریتم‌های چکیده ساز رایج عبارت‌اند از:

خانواده MD[۱۸]: MD6,MD5,MD4,MD2

خانواده SHA[۱۹]: SHA-0,SHA-1,SHA-2,SHA-3

در این میان از استفاده از SHA-0,SHA-1,MD4,MD2 و MD6 به دلیل ضعف امنیتی باید پرهیز کرد، همچنین MD5 که یکی از فراگیرترین چکیده سازهای حال حاضر است نیز به دلیل ضعف امنیتی کنار گذاشته‌شده است.

در حال حاضر SHA-3 یکی از ایمن‌ترین چکیده سازهای حاضر است.

### ۲-۶-۲ رمزنگاری با کلید متقارن[۲۰]

در این روش از رمزنگاری، ورودی با استفاده از یک یا چند کلید رمزنگاری‌شده و خروجی با استفاده از همان کلید (ها) به ورودی اولیه بازمی‌گردد. این ویژگی این امکان را فراهم می‌کند تا اطلاعاتی همانند

---

[۱۷] Hash

[۱۸] Message-Digest Algorithm

[۱۹] Secure Hash Algorithm

[۲۰] Symmetric Key

مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

اطلاعات اتصال به بانک اطلاعاتی[۲۱] را درون یک پرونده به‌صورت رمزنگاری‌شده ذخیره کرده تا از در صورت باز شدن آن پرونده توسط شخص تأیید نشده، اطلاعات ورود به بانک اطلاعاتی قابل‌رؤیت نباشد و در زمان فراخوانی توسط برنامه آن را رمزگشایی کنیم.

چند نمونه از الگوریتم‌های رایج رمزنگاری با کلید متقارن عبارت‌اند از:

RC4[۲۲] ,AES[۲۳] ,DES[۲۴] ,3DES[۲۵]

که به‌دلیل وجود ضعف امنیتی در این الگوریتم‌ها، در حال حاضر AES با سایز کلید ۲۵۶ بیت از ایمن‌ترین راهکارها است.

### ۳-۶-۲ رمزنگاری با کلید عمومی[۲۶] (نامتقارن[۲۷])

این نوع رمزنگاری با استفاده از یک کلید عملیات رمزنگاری را انجام داده و با استفاده از یک کلید دیگر عملیات رمزگشایی را انجام می‌دهد.

عمدتاً از کلیک عمومی که در دسترس عام است برای عملیات رمزنگاری و از کلید خصوصی که در دسترس برنامه و افراد تأییدشده است برای عملیات رمزگشایی استفاده می‌شود.

این نوع رمزنگاری عمدتاً در ارتباط بین سرویس‌دهنده[۲۸] و سرویس‌گیرنده[۲۹] استفاده می‌شود. همچنین در پروانه[۳۰] برنامه‌ها و تأیید اطلاعات ارسالی نیز کاربرد دارد و برای امضای دیجیتال نیز به کار می‌رود.

---

Database[۲۱]
Rivest Cipher 4[۲۲]
Advanced Encryption Standard[۲۳]
Data Encryption Algorithm[۲۴]
Triple Data Encryption Algorithm[۲۵]
Public Key[۲۶]
Asymmetric[۲۷]
Server[۲۸]
Client[۲۹]
License[۳۰]

مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه‌ای

ویژگی این رمزنگاری این است که با اعلان عمومی کلید عمومی در برنامه، خطری داده‌ها را تهدید نکرده چراکه کلید خصوصی محفوظ است. هر کلید عمومی فقط و فقط یک کلید خصوصی دارد و هر کلید خصوصی فقط و فقط یک کلید عمومی دارد. درواقع کلید عمومی و خصوصی یک رمزنگاری نامتقارن منحصربه‌فرد می‌باشند و با یکدیگر رابطه یک‌به‌یک دارند.

رایج‌ترین الگوریتم‌های رمزنگاری نامتقارن عبارت‌اند از: RSA ,DSA[۳۱]/DSS[۳۲]

بحث رمزنگاری از پیچیده‌ترین بحث‌ها در رایانه است و شامل انواع دیگر و زیر دسته‌های مختلف می‌شود و مطالب ذکرشده مطالب کلی موردنیاز در بحث برنامه‌نویسی تدافعی و ایمن می‌باشند.

## ۷-۲ کدگذاری دودویی به متن[۳۳]

کدگذاری دودویی به متن درواقع نوعی رمزنگاری بازگشت‌پذیر بدون کلید است.

در این روش باینری‌ها به کاراکترهای خوانا تبدیل شده و امکان ذخیره‌سازی آن‌ها به‌عنوان پرونده متنی فراهم می‌شود.

در کدگذاری دودویی به متن امکان بروز تداخل[۳۴] وجود نداشته و سایز خروجی بسته به سایز ورودی متفاوت است.

خانواده Base از رایج‌ترین این دسته می‌باشند. Base85, Base64, Base58, Base32, Base16 Base91 اعضای این خانواده می‌باشند.

Base64 فراگیرترین عضو این خانواده است که داده‌های باینری را به‌صورت کاراکترهای ASCII نمایان می‌کند.

برای انجام کدگذاری دودویی به متن می‌توان از کتابخانه‌های معرفی‌شده در بخش رمزنگاری استفاده کرد.

---

[۳۱] Digital Signature Algorithm

[۳۲] Digital Signature Standard

[۳۳] Binary-to-Text Encode

[۳۴] Collision

# ۳ بررسی ایمن داده‌ها و اشیاء

## ۳-۱ مدیریت مقدار و نوع داده‌های بنیادی[۳۵]

مهم‌ترین عناصر یک برنامه داده‌های ورودی و خروجی آن برنامه می‌باشند. بدیهی است که در برنامه‌های کاربردی ورود اطلاعات توسط کاربر امری بسیار مرسوم است. با توجه به تولید و مدیریت داده‌های خروجی توسط برنامه، نیاز است تا با مدیریت نوع داده ورودی از صحت نوع داده ورودی اطمینان پیدا کنیم. داده‌های بنیادی آن دسته از داده‌ها می‌باشند که خود مشتق و یا ترکیبی از نوع داده دیگری نمی‌باشند.

## ۳-۲ مدیریت مقدار و نوع داده‌های بنیادی: مقداردهی اولیه[۳۶]

مقداردهی اولیه متغیرهایی با نوع داده بنیادی امری ضروری است چراکه در زمان اختصاص حافظه به متغیر ایجادشده فضای اختصاص‌یافته پاک‌سازی نمی‌شود. البته در برخی از کامپایلرها همانند Microsoft Visual C++ Compiler مقداردهی اولیه در برخی از متغیرهای با نوع بنیادی به‌صورت خودکار انجام می‌پذیرد اما این امر نباید باعث عدم مقداردهی اولیه شود چراکه اصول برنامه‌نویسی ایمن باید جدا از امکانات کامپایلر پیاده‌سازی شوند و در مواقع لزوم از امکانات اضافی کامپایلرها که در پیش‌فرض زبان برنامه‌نویسی نیست استفاده گردد.

قطعه کد زیر نمونه‌ای از تعریف تغییر از نوع داده بنیادی بدون مقداردهی اولیه است.

```
int main (void)
{
      int a;
      float b;
      char c;
      bool d;
      return 0;
}
```

[۳۵] Fundamental Data Types

[۳۶] Initialization

مقادیر متغیرهای این کد بعد از اجرا بدین‌صورت است:

| Name | Value | Type |
|---|---|---|
| a | 6618996 | int |
| b | 9.275e-039#DEN | float |
| c | 116 't' | char |
| d | true (116) | bool |

مقداردهی اولیه متغیرها یا باید برحسب نیاز در همان ابتدا صورت گیرد و یا در صورت عدم نیاز به وجود مقدار اولیه خاص، مقداردهی با استفاده از تابع همان نوع داده بنیادی انجام شود.

قطعه کد زیر نمونه‌ای از مقداردهی اولیه صحیح با استفاده از تابع نوع داده بنیادی است.

```
int main (void)
{
        int a = int();
        float b = float();
        char c = char();
        bool d = bool();
        return 0;
}
```

خروجی قطعه کد بالا به‌صورت زیر است:

| Name | Value | Type |
|---|---|---|
| a | 0 | int |
| b | 0.000000000 | float |
| c | 0 '\0' | char |
| d | false | bool |

## ۳-۳ مدیریت مقدار و نوع داده‌های بنیادی: مقداردهی اولیه آرایه‌ها[۳۷]

همانند متغیرهای با نوع داده بنیادی، آرایه‌های با نوع داده بنیادی نیز باید در زمان تعریف مقداردهی اولیه داشته باشند چراکه در زمان اختصاص حافظه به آرایه نیز اطلاعات موجود در آن بخش از حافظه پاک نخواهد شد.

---

[۳۷] Array Initialization

قطعه کد زیر نمونه‌ای از تعریف آرایه بدون مقداردهی اولیه است:

```
int main (void)
{
        int a[5];
        float b[5];
        char c[5];
        bool d[5];
        return 0;
}
```

خروجی قطعه کد بالا به‌صورت زیر است:

| Name | Value | Type |
|---|---|---|
| a | 0x00e9f9ec {15333948, 16008761, 1, 17875368, 17872688} | int[5] |
| b | 0x00e9f9ec {2.14874378e-038, 2.24330522e-038, 1.401e-045#DEN, 2.65875644e-038, 2.65800535e-038} | float[5] |
| c | 0x00e9f9ec "<úé" | char[5] |
| d | 0x00e9f9ec {true (60), true (250), true (233), false, true (57)} | bool[5] |

در صورت عدم نیاز به مقداردهی اولیه آرایه‌ها با مقداری خاص، آرایه‌ها بر طبق کد زیر و با اپراتورهای براکت مجعد[۳۸] باز و بسته باید مقداردهی شوند.

قطعه کد زیر به‌صورت صحیح آرایه‌ها را مقداردهی می‌کند:

```
int main (void)
{
        int a[5] = {};
        float b[5] = {};
        char c[5] = {};
        bool d[5] = {};
        return 0;
}
```

خروجی قطعه کد بالا به‌صورت زیر است:

| Name | Value | Type |
|---|---|---|
| a | 0x009cfa40 {0, 0, 0, 0, 0} | int[5] |
| b | 0x009cfa54 {0.000000000, 0.000000000, 0.000000000, 0.000000000, 0.000000000} | float[5] |
| c | 0x009cfa34 "" | char[5] |
| d | 0x009cfa39 {false, false, false, false, false} | bool[5] |

[۳۸] Curly Brackets, Curly Braces

# ۴ بررسی ایمن داده‌ها و اشیاء

## ۴-۱ قالب: تغییر قالب پایدار[۳۹]

یکی از کلیدواژه‌های کاربردی در زبان C/C++ کلیدواژه const است که باعث پایدار بودن و غیر قابل‌تغییر بودن شیء می‌شود. با استفاده از قالب const_cast می‌توان صفات const,volatile,_unaligend[۴۰] را از شیء جدا کرده و آن را در ساختاری دیگر قرار داد.

یکی از آسیب‌پذیری‌هایی که در این فرایند ممکن است رخ دهد و به رفتار تعریف‌نشده منجر شود، اشاره یک اشاره‌گر غیر پایدار به مکانی در حافظه باشد که یک شیء یا اشاره‌گر پایدار به آن اشاره می‌کند. در این صورت با تغییر مقدار آن اشاره‌گر، رفتار غیرمتعارف رخ می‌دهد.

قطعه کد نمونه‌ای از این رفتار است:

```
int main (void)
{
        const int a = 1;
        int* = const_cast<int*>(&a);
        *b = 2;
        return 0;
}
```

متغیر a از نوع عدد صحیح و باصفت پایدار تعریف‌شده و عدد ۱ مقداردهی شده است. بدیهی است که مقدار a قابل‌تغییر نیست. سپس اشاره‌گر b از نوع عدد صحیح به مکان a در حافظه اشاره می‌کند. در قدم بعد مقدار حافظه اشاره‌شده توسط اشاره‌گر b با عدد ۲ مقداردهی می‌شود. نتیجه اجرای کد بالا به‌صورت زیر است:

| Name | Value | Type |
|---|---|---|
| a | 2 | const int |
| b | 0x0018f9b0 {2} | int * |

---

[۳۹] Const Cast

[۴۰] Attributes

همان‌گونه که مشاهده می‌شود باوجوداینکه a یک عدد صحیح پایدار است اما مقدار آن به ۲ تغییر کرده است.

در استفاده از const_cast باید دقت داشت که در هیچ شرایطی، حالت فوق رخ نداده و همچنین تا حد امکان از استفاده const_cast دوری کرد.

مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

# ۵ اشاره‌گر[۴۱]

## ۵-۱ مقداردهی اولیه

اشاره‌گرها در زمان تعریف باید مقداردهی اولیه داشته باشند، در غیر این صورت یک اشاره‌گر رام نشده می‌باشد.

قطعه کد زیر نمایان گر تعریف اشاره‌گر بدون مقداردهی اولیه است:

```
int main (void)
{
        int *a;
        float *b;
        char *c;
        bool *d;
        return 0;
}
```

خروجی قطعه کد بالا به‌صورت زیر است:

| Name | Value | Type |
|---|---|---|
| a | 0x0103f998 {0x0103f9a0} | int * |
| b | 0x0103f998 {2.42399971e-038} | float * |
| c | 0x0103f998 " ù\x3\x1=H\xf\x1´ù\x3\x1D7" | char * |
| d | 0x0103f998 {true (0xa0)} | bool * |

چنانچه اشاره‌گر نیاز به مقداردهی خاص اولیه‌ای نداشته باشد، با استفاده از ایجاد یک شیء جدید از نوع داده‌ای که اشاره‌گر از آن تعریف‌شده، مقداردهی اولیه انجام می‌شود.

قطعه کد زیر نحوه مقداردهی اولیه اشاره‌گر را به نمایش می‌گذارد:

```
int main (void)
{
        int *a = new int();
        float *b = new float();
        char *c = new char();
        bool *d = new bool();
        return 0;
}
```

[۴۱] Pointer

خروجی قطعه کد بالا به‌صورت زیر است:

| Name | Value | Type |
|---|---|---|
| a | 0x008dc230 {0x00000000} | int * |
| b | 0x008d7e10 {0.000000000} | float * |
| c | 0x008d7e50 "" | char * |
| d | 0x008d4dc0 {false} | bool * |

دقت داشته باشید که با انجام این عمل، تخصیص حافظه[۴۲] صورت می‌گیرد و باید آزادسازی حافظه[۴۳] انجام شود.

## ۵-۲ اشاره‌گر تهی[۴۴]

اشاره‌گر تهی به اشاره‌گری گفته می‌شود که به مکان خاصی در حافظه اشاره نمی‌کند.

نحوه تعریف صحیح اشاره‌گر تهی به‌صورت زیر است:

```
int main (void)
{
        char *ptr = nullptr;
        return 0;
}
```

nullptr نوع داده معرفی‌شده در استاندارد C++11 برای اشاره‌گر تهی است.

در استانداردهای قبلی از null برای این منظور استفاده می‌شد. استفاده از عدد ۰ که درگذشته مورداستفاده قرار می‌گرفت در استاندارد جدید به‌عنوان تهی محسوب نشده و عدد صحیح است لذا باید از استفاده از آن اجتناب کرد.

خروجی قطعه بالا به‌صورت زیر است:

| Name | Value |
|---|---|
| ptr | 0x00000000 <NULL> |

مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

Memory Allocation[۴۲]

Memory Deallocation[۴۳]

Null Pointer[۴۴]

که مقدار ptr آدرس اشاره‌شده به مکانی در حافظه است که در حال حاضر به هیچ مکانی اشاره نمی‌کند.

یکی از کاربردهای رایج اشاره‌گر تهی مشخص کردن انتهای لیست پیوندی است اما باید دقت داشت که هرگونه ارجاع به اشاره‌گر تهی یک ریسک امنیتی محسوب می‌شود. آسیب‌پذیری ارجاع به اشاره‌گر تهی[۴۵] زمانی رخ می‌دهد که ارجاعی به اشاره‌گری که مقدار آن تهی است داده می‌شود و درنتیجه به دلیل عدم وجود آدرس در آن اشاره‌گر، برنامه با ریسک امنیتی مواجه می‌شود.

قطعه کد زیر نمونه‌ای از آسیب‌پذیری ارجاع به اشاره‌گر تهی است:

```
int main (void)
{
        int *a = nullptr;
        int *b;
        *b = *a;
        return 0;
}
```

در قطعه کد بالا مقدار عددی موجود در آدرس اشاره‌شده توسط اشاره‌گر b برابر مقدار عددی موجود در آدرس اشاره‌شده توسط اشاره‌گر a قرار می‌گیرد اما به دلیل عدم اشاره اشاره‌گر a به قسمتی از حافظه، مقداری برای بازگردانی و قرارگیری به‌عنوان مقدار عددی آدرس اشاره‌شده توسط اشاره‌گر b وجود نداشته و آسیب‌پذیری ارجاع به اشاره‌گر تهی پدید می‌آید.

برای برطرف سازی این آسیب‌پذیری کافی است قبل از ارجاع به اشاره‌گر، تهی بودن آن مورد ارزیابی قرار گیرد.

قطعه کد نحوه جلوگیری از بروز این آسیب‌پذیری را به نمایش می‌گذارد:

```
int main (void)
{
        int *a = nullptr;
        int *b;
        if (a != nullptr)
                *b = *a;
        return 0;
}
```

همچنین اشاره‌گر تهی نباید به char_traits::length و توابعی که از آن استفاده می‌کنند، ارسال شود.

---

[۴۵] NullPointer Dereference

## ۳-۵ اشاره‌گر معلق[۴۶]

اشاره‌گر معلق به اشاره‌گری گفته می‌شود که به آدرس شیء خاصی در حافظه اشاره نمی‌کند اما این آدرس در زمان گذشته و یا شرایطی خاص به شیء خاصی اشاره می‌کند.

اشاره‌گر معلق عمدتاً به دلیل استفاده از اشاره‌گر بعد از پاک‌سازی شیء و یا بازگرداندن آدرس متغیر محلی اختصاص داده‌شده به‌واسطه Stack به وجود می‌آید.

قطعه کد زیر نمونه‌ای از اشاره‌گر معلق به دلیل استفاده از آدرس شیء از بین رفته است:

```
int main (void)
{
        char *a = new char[10];
        *a = char()
        delete[] a;
        return 0;
}
```

قبل از اجرای delete مقدار اشاره‌گر به‌صورت زیر می‌باشند:

| Name | Value | Type |
|---|---|---|
| a | 0x011c7e10 "" | char * |

```
011C7E10 00 CD
011C7E12 CD CD
011C7E14 CD CD
011C7E16 CD CD
011C7E18 CD CD
```

در انتهای کد مقدار اشاره‌گر به‌صورت زیر می‌باشند:

[۴۶] Dangling Pointer

| Name | Value | Type |
|---|---|---|
| a | 0x011c7e10 "îþîþîþîþîþîþîþî | char * |

```
011C7E10 EE
011C7E11 ??
011C7E12 EE
011C7E13 ??
011C7E14 EE
011C7E15 ??
011C7E16 EE
011C7E17 ??
011C7E18 EE
011C7E19 ??
```

بعد از اجرای delete، اشاره‌گر a یک اشاره‌گر معلق است.

به حالتی که یک اشاره‌گر بعد از آزادسازی به حالت معلق درآمده اما مجدداً مورداستفاده قرار می‌گیرد، استفاده بعد از آزادسازی[۴۷] گفته می‌شود.

قطعه کد زیر نمونه‌ای از اشاره‌گر معلق به دلیل بازگرداندن آدرس تغییر محلی است:

```
int *ptrFunc(void)
{
	int a = int();
	return &a;
}

int main (void)
{
	int *ptr = ptrFunc();
	return 0;
}
```

در قطعه کد بالا آدرس متغیر a که یک متغیر محلی است بازگردانی می‌شود و درنتیجه این عمل، اشاره‌گر ptr یک اشاره‌گر معلق است.

اشاره‌گرهای معلق نیز یکی از ریسک‌های قابل‌توجه امنیتی محسوب می‌شود که باعث افشای اطلاعات، تغییر سطح دسترسی، استفاده از اشاره‌گر برای فراخوانی تابع مجازی[۴۸] و اجرای کد دلخواه مهاجم با استفاده از بازنویسی اشاره‌گر جدول مجازی متد[۴۹] می‌شود.

---

Use-After-Free[۴۷]

Virtual Function[۴۸]

Virtual Method Table (vtable)[۴۹]

مرکز مدیریت امداد و هماهنگی عملیات واحدهای رایانه ای

قطعه کد زیر چگونگی مدیریت و جلوگیری از اشاره‌گر معلق را به نمایش می‌گذارد:

```
int main (void)
{
        char *a = new char[10];
        *a = char()
        delete[] a;
        a = NULL;
        return 0;
}
```

در قطعه کد بالا استفاده از NULL باعث از بین رفتن اشاره‌گر معلق می‌شود. به این روش تهی سازی اشاره‌گر معلق[۵۰] گفته می‌شود. در استاندارد جدید از null_ptr به‌جای NULL استفاده می‌گردد.

## ۵-۴ اشاره‌گر رام نشده[۵۱]

اشاره‌گر رام نشده به اشاره‌گری گفته می‌شود که مقدار آن به یک شیء معتبر در حافظه اشاره نمی‌کند، این دسته از اشاره‌گرها به دلیل عدم مقداردهی اولیه اشاره‌گر به وجود می‌آیند. عدم مقداردهی اولیه اشاره‌گر عمدتاً یا خطای برنامه‌نویس بوده و یا عدم اجرای کد نوشته‌شده برای مقداردهی اولیه اشاره‌گر به دلیل وجود دستوراتی همانند دستورات شرطی و پرش است که در شرایطی باعث عدم اجرای کد مقداردهی می‌شوند.

قطعه کد زیر نمونه‌ای از یک اشاره‌گر رام نشده است:

```
int main (void)
{
        char *ptr;
        return 0;
}
```

اشاره‌گرهای رام نشده در برخی از تعاریف به‌عنوان زیرمجموعه‌ای از اشاره‌گرهای معلق معرفی می‌شوند.

برای جلوگیری از وجود اشاره‌گرهای رام نشده در برنامه باید اشاره‌گر در هر شرایطی دارای مقدار معتبر باشد که این مقدار می‌تواند از طریق مقداردهی اولیه و یا شروط و حلقه‌هایی که به‌طورقطع اشاره‌گر را مقداردهی می‌کنند انجام شود.

---

Dangling Pointer Nullification[۵۰]

Wild Pointer[۵۱]

## ۵-۵ اشاره‌گر هوشمند[۵۲]

اشاره‌گرهای هوشمند یک نوع داده انتزاعی[۵۳] می‌باشند که نقش اشاره‌گر را شبیه‌سازی می‌کنند.

یکی از اهداف اصلی استفاده از اشاره‌گرهای هوشمند برای به حداقل رساندن آسیب‌پذیری‌های به وجود آمده ناشی از استفاده غیر ایمن از اشاره‌گرهاست.

اشاره‌گرهای هوشمند ویژگی‌هایی همانند مدیریت حافظه[۵۴] و بررسی کران[۵۵]ها را در اختیار برنامه‌نویس قرار می‌دهند که این ویژگی‌ها به‌صورت خودکار و بدون نیاز به برنامه‌نویسی اعمال می‌شوند.

اشاره‌گرهای هوشمند نیازمند منابع بیشتری نسبت به اشاره‌گرهای معمولی هستند و عملکرد کندتری به نسبت اشاره‌گرهای معمولی دارند لذا استفاده از آن‌ها در برنامه‌هایی که نیازمند بهره‌برداری از حداکثر منابع و سرعت‌بالا می‌باشند، توصیه نمی‌گردد.

اشاره‌گرهای هوشمند اصلی در هدر memory قرار دارند.

۳ دسته اصلی اشاره‌گرهای هوشمند به‌صورت زیر می‌باشند:

۱- unique_ptr

این دسته از اشاره‌گرها در حالت کلی جایگزین اشاره‌گرهای سنتی می‌باشند. این نوع اشاره‌گر تنها یک مالک داشته و جایگزین auto_ptr در استانداردهای قدیمی‌تر است.

۲- shared_ptr

این نوع اشاره‌گر، اشاره‌گر شمارشی ارجاع داده‌شده[۵۶] است و امکان تخصیص یک اشاره‌گر به چند مالک را داراست. اشاره‌گر تنها در صورتی از بین می‌رود که تمامی مالکان آن از مالکیتش صرف‌نظر کرده و یا به خارج از محدوده بروند.

---

[۵۲] Smart Pointer
[۵۳] Abstract Data Type
[۵۴] Memory Management
[۵۵] Bound Checking
[۵۶] Reference-Counted

مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

۳- weak_ptr

این نوع اشاره‌گر در ترکیب با shared_ptr استفاده می‌شود و برای دسترسی به شیء ای که تحت مالکیت یک یا چند shared_ptr است به کار می‌رود با این تفاوت که در شمارش ارجاع‌ها شرکت داده نمی‌شود. این اشاره‌گر درزمانی استفاده می‌شود که هدف مشاهده یک شیء است اما نیازی به نگه‌داشتن موجودیت آن نیست. همچنین از این اشاره‌گر برای شکستن ارتباط حلقه‌ای بین shared_ptr ها نیز استفاده می‌شود.

قطعه کد زیر نمونه‌ای عملکرد اشاره‌گر هوشمند در مقایسه با اشاره‌گر معمولی است:

```
int main (void)
#include <memory>

int main (void)
{
	std::unique_ptr<int> a(new int());
	int *b = new int();
	*a = 1;
	*b = 1;
	a.release();
	delete[] b;
	return 0;
}
```

در قطعه کد بالا، ابتدا اشاره‌گر a از نوع unique_ptr و عدد صحیح و سپس اشاره‌گر b از نوع عدد صحیح ساخته می‌شوند و سپس مقدار ۱ به‌عنوان مقدار رقمی آن‌ها، تخصیص داده می‌شود.

در قدم بعد هر دو اشاره‌گر آزادسازی می‌شوند.

خروجی قطعه کد بالا به‌صورت زیر است:

| Name | Value |
|---|---|
| a | empty |
| b | 0x00605100 {-17891602} |

بعد از آزادسازی، اشاره‌گر a به‌طور کامل خالی‌شده اما اشاره‌گر b به اشاره‌گر معلق تبدیل می‌شود.

باید دقت داشت که اشاره‌گرهای هوشمند std::unique_ptr و std::shared_ptr، در زمان ایجاد در صورت مقداردهی آن‌ها توسط اشاره‌گر، مقدار اشاره‌گر را از آن خود می‌کنند.

در این میان std::unique_ptr در زمان تخریب، آزادسازی و بازنشانی مالکیت مقدار را بازگردانده اما std::shared_ptr باعث نابودی آن می‌شود.

مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

ساخت یک shared_ptr از روی یک shared_ptr دیگر باعث بروز این حالت نشده و در زمان ایجاد، یک نسخه از روی نسخه اصلی برداشته‌شده و بین این دو یک رابطه ایجاد می‌شود.

قطعه کد زیر نمونه‌ای از این موضوع است:

```
int *i = new int();
std::shared_ptr<int> p1(i);
std::shared_ptr<int> p2(i);
```

در این حالت چنانچه که p2 تخریب شود مقدار اشاره‌گر i نیز پاک می‌شود و درصورتی‌که بعدازآن p1 نیز تخریب شود، آسیب‌پذیر آزادسازی مضاعف[۵۷] رخ خواهد داد چراکه i وجود ندارد.

قطعه کد زیر نمونه اصلاح‌شده بالا با توجه به ویژگی نسخه‌برداری share_ptr است:

```
std::shared_ptr<int> p1 = std::make_shared<int>();
std::shared_ptr<int> p2(p1);
```

در این قطعه کد با استفاده از std::make_shared یک اشاره‌گر از نوع عدد صحیح تعریف‌شده و به p1 مقداردهی می‌شود و سپس p2 از روی p1 نسخه‌برداری می‌کند.

مرکز رشد پژوهش امداد و هماهنگی عملیات رخدادهای رایانه ای

[۵۷] Double Free

# ۶ حافظه

منظور از حافظه در این مستند، حافظه اصلی است.

## ۶-۱ نشت حافظه[۵۸]

نشت حافظه زمانی اتفاق می‌افتد که حافظه اختصاص داده‌شده به شیء ای خاص بعد از اتمام نیاز به آن شیء آزادسازی نشود. در این حالت آن قسمت از حافظه در اشغال برنامه است درحالی‌که استفاده‌ای از آن نمی‌شود. همچنین ممکن است آن قسمت از حافظه توسط کد اجرایی غیرقابل‌دسترس شود. وجود نشت حافظه ممکن است باعث پر شدن حافظه و بروز مشکلاتی همانند کندی سیستم، ازکارافتادن و کندی برنامه شود.

قطعه کد زیر نمونه‌ای از نشت حافظه است:

```
void memory_leak(void)
{
	int *a = new int();
	*a = 1;
}
```

در قطعه کد بالا ابتدا حافظه‌ای اختصاص داده می‌شود و a اشاره‌گر آدرس آن بخش تخصیص‌یافته است. حافظه اشاره‌شده توسط a بعد از اتمام تابع memory_leak آزادسازی نشده و باعث بروز نشت حافظه می‌شود. همچنین اشاره‌گر a به یک اشاره‌گر خارج از محدوده[۵۹] تبدیل می‌شود.

---

Memory Leak[۵۸]

Out Of Scope[۵۹]

قطعه کد زیر نمونه‌ای از تخصیص حافظه با استفاده از تابع malloc() است:

```
void memory_leak(void)
{
	int *a = (int*)std::malloc(sizeof(int));
	*a = 1;
}
```

برای جلوگیری از بروز نشت حافظه می‌توان از اشاره‌گرهای هوشمند استفاده کرد. همچنین باید از free برای آزادسازی حافظه تخصیص‌یافته توسط malloc() و delete برای آزادسازی حافظه تخصیص‌یافته توسط new استفاده کرد.

قطعه کدهای زیر نمونه‌های اصلاح‌شده دو قطعه کد بالا می‌باشند:

```
void memory_leak(void)
{
	int *a = new int();
	*a = 1;
	delete[] a;
}
```

```
void memory_leak(void)
{
	int *a = (int*)std::malloc(sizeof(int));
	*a = 1;
	free(a);
}
```

تخصیص دهنده[60]ها و آزادکننده[61]ها در زبان C++ به‌صورت زیر می‌باشد:

مرکز مدیریت امداد و هماهنگی عملیات خدادهای رایانه ای

[60] Allocator

[61] Deallocator

| Allocator | Deallocator |
|---|---|
| operator new()/new | operator delete()/delete |
| operator new[]()/new[] | operator delete[]()/delete[] |
| placement operator new() | N/A |
| allocator<T>::allocate() | allocator<T>::deallocate() |
| std::malloc(), std::calloc(), std::realloc() | std::free() |
| std::get_temporary_buffer() | std::return_temporary_buffer() |

[62]

تخصیص دهنده‌ها و آزادکننده‌ها در سیستم‌عامل ویندوز به‌صورت زیر می‌باشند:

مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

[62] https://securecoding.cert.org/confluence/display/cplusplus/MEM51-CPP.+Properly+deallocate+dynamically+allocated+resources

| Allocator | Deallocator |
|---|---|
| malloc() | free() |
| realloc() | free() |
| LocalAlloc() | LocalFree() |
| LocalReAlloc() | LocalFree() |
| GlobalAlloc() | GlobalFree() |
| GlobalReAlloc() | GlobalFree() |
| VirtualAlloc() | VirtualFree() |
| VirtualAllocEx() | VirtualFreeEx() |
| VirtualAllocExNuma() | VirtualFreeEx() |
| AllocateUserPhysicalPages() | FreeUserPhysicalPages() |
| AllocateUserPhysicalPagesNuma() | FreeUserPhysicalPages() |
| HeapAlloc() | HeapFree() |
| HeapReAlloc() | HeapFree() |

## ۶-۲ آزادسازی مضاعف

آزادسازی مضاعف زمانی صورت می‌گیرد که حافظه تخصیص‌یافته به یک اشاره‌گر در سیر برنامه ۲ بار به‌صورت متوالی آزادسازی گردد.

این آسیب‌پذیری ممکن است باعث خرابی حافظه، فراهم آوردن امکان اجرای کد مخرب و ازکارافتادن برنامه شود.

قطعه کد زیر نمونه‌ای از آسیب‌پذیری آزادسازی مضاعف است:

```
#include <iostream>
```

```
int main(void)
{
	int *a = (int*)std::malloc(sizeof(int));
	std::cin >> *a;
	int b = *a;
	if (b % 2 == 0)
	{
		printf("Division by 2 is 0");
		free(a);
	}
	if (b % 3 == 0)
	{
		printf("Division by 3 is 0");
		free(a);
	}
	return 0;
}
```

چنانچه ورودی این قطعه کد عددی باشد که مضربی از ۲ و ۳ باشد، آزادسازی مضاعف انجام می‌گیرد.

برای جلوگیری از بروز این آسیب‌پذیری باید از هر شرایطی که باعث ایجاد آزادسازی مضاعف می‌شود پرهیز کرد و از یک دستور آزادسازی به‌گونه‌ای استفاده کرد که تمامی حالات را پوشش دهد. همچنین استفاده از اشاره‌گر هوشمند نیز باعث جلوگیری از بروز این آسیب‌پذیری می‌شود.

قطعه کد زیر نمونه اصلاح‌شده قطعه کد بالا است:

```
#include <iostream>

int main(void)
{
	int *a = (int*)std::malloc(sizeof(int));
	std::cin >> *a;
	int b = *a;
	if (b % 2 == 0)
		printf("Division by 2 is 0");
	if (b % 3 == 0)
		printf("Division by 3 is 0");
	free(a);
	return 0;
}
```

مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه‌ای

## ۳-۶ آزادسازی اشاره‌گر تهی

آزادسازی اشاره‌گر تهی زمانی رخ می‌دهد که یک اشاره‌گر از ابتدا تهی بوده و یا درروند اجرایی برنامه تهی گردد و سپس آزادسازی شود.

در استانداردهای زبان C/C++ از ویرایش C99 به بعد در صورت ارجاع یک اشاره‌گر تهی به تابع آزادسازی، عملی صورت نگرفته و باعث بروز خطا نمی‌شود، بااین‌وجود آزادسازی اشاره‌گر تهی هنوز هم یکی از مباحث برنامه‌نویسی تدافعی و ایمن است.

قطعه کد زیر نمونه‌ای از آزادسازی اشاره‌گر تهی است:

```
int main(void)
{
        char *ptr = nullptr;
        free(ptr);
        return 0;
}
```

برای جلوگیری از بروز این آسیب‌پذیری، می‌توان از یک شرط برای بررسی تهی بودن اشاره‌گر استفاده کرد.

دقت داشته باشید که برای جلوگیری از بروز اشاره‌گر معلق، اشاره‌گر بعد از آزادسازی باید تهی گردد.

قطعه کد زیر، نسخه ایمن شده قطعه کد بالا است:

```
int main(void)
{
        char *ptr = nullptr;
        if (ptr != nullptr)
        {
                free(ptr);
                ptr = nullptr;
        }
        return 0;
}
```

## ٤-٦ پاک‌سازی ایمن، صفر سازی[٦٣] حافظه

یکی از روش‌های اطمینان از عدم وجود هرگونه اطلاعات بازمانده از یک شیء، پاک‌سازی ایمن حافظه تخصیص‌یافته به آن شیء با استفاده از صفر سازی حافظه است.

باید دقت داشت که در این روش ابتدا حافظه تخصیص‌یافته به شیء باید آزادسازی شده و سپس صفر سازی حافظه بر روی آن انجام شود.

تابع SecureZeroMemory() در سیستم‌عامل ویندوز واقع در هدر Windows.h برای این منظور استفاده می‌شود.

ساختار این تابع به‌صورت زیر است:

```
PVOID SecureZeroMemory(
  _In_ PVOID  ptr,
  _In_ SIZE_T cnt
);
```

ورودی این تابع، اشاره‌گر آدرس شروع‌کننده و سایز موردنظر برای صفر سازی، به بایت است.

قطعه کد زیر نحوه استفاده از این تابع را به نمایش می‌گذارد:

```
#include <windows.h>

int main(void)
{
	char *a = "Password";
	SecureZeroMemory(a,8);
	return 0;
}
```

در قطعه کد بالا ۸ بایت عبارت Password صفر سازی می‌شود. به پایان رسانده تهی نیز خود صفر است.

خروجی قطعه کد بالا قبل از اجرای تابع SecureZeroMemory به‌صورت زیر است:

| Name | Value |
|---|---|
| a | 0x00a39000 "Password" |
| | 80 'P' |

[٦٣] Zero-Fill

```
00A39000 50
00A39001 61
00A39002 73 73
00A39004 77 6F
00A39006 72 64
```

خروجی قطعه در انتها به‌صورت زیر است:

| Name | Value |
|---|---|
| a | 0x00a39000 "" |
| | 0 '\0' |

```
00A39000 00 00
00A39002 00 00
00A39004 00 00
00A39006 00 00
```

## ۶-۵ فاش شدن اطلاعات[۶۴]

اطلاعات درون حافظه به‌صورت متن ساده[۶۵] قرار می‌گیرند که درنتیجه امکان فراخوانی و دسترسی به آن توسط عموم امکان‌پذیر می‌شود. بلوک‌های حافظه مربوط به برنامه خوانده می‌شود و سپس اطلاعات ذخیره‌شده در آن استخراج می‌گردد.

استخراج این اطلاعات زمانی اهمیت پیدا می‌کند که شما اطلاعات بااهمیتی را درون حافظه ذخیره می‌کنید.

قطعه کد زیرنمایان گر این موضوع است:

```
#include <iostream>
#include <string>

int main(void)
{
        std::string input;
        std::cin >> input;
        return 0;
}
```

حال بعد از اجرای این برنامه، ورود کلمه test و خواندن حافظه برنامه، خروجی به‌صورت زیر است:

---

Information Leakage[۶۴]

Plain Text[۶۵]

| Name | Value |
|---|---|
| input | "test" |

مقدار input به‌صورت متن ساده، درون حافظه قرارگرفته است.

برای جلوگیری از بروز این امر، باید از رمزنگاری‌های قابل بازگشت برای ذخیره اطلاعات حساس در حافظه استفاده کرد.

یکی از روش‌های مرسوم استفاده از XOR است.

قطعه کد زیر نمونه‌ای از رمزنگاری اطلاعات با استفاده از XOR است:

```
#include <iostream>
#include <string>

int main(void)
{
	std::string input;
	std::cin >> input;
	for (int i = 0; i < input.size(); i++)
		input[i] = input[i] ^ 1;
	return 0;
}
```

خروجی قطعه کد بالا در صورت ورود test به‌صورت زیر است:

| Name | Value |
|---|---|
| input | "udru" |

همان‌طور که مشخص است، محتویات input به کاراکترهای نامفهوم تبدیل‌شده‌اند که در اصل دربردارنده همان کلمه test هستند.

برای بازگردانی کلمه کافی است عمل XOR را تکرار کنیم.

قطعه کد زیر ابتدا ورودی را رمزنگاری کرده و سپس واژه رمزگشایی‌شده را درون متغیر dec قرار می‌دهد:

```
#include <iostream>
#include <string>

int main(void)
{
	std::string input;
	std::string dec = "";
	std::cin >> input;
	for (int i = 0; i < input.size(); i++)
		input[i] = input[i] ^ 1;
	for (int j = 0; j < input.size(); j++)
		dec += input[j] ^ 1;
	return 0;
}
```

خروجی قطعه کد بالا در صورت ورود test به‌صورت زیر است:

| Name | Value |
|---|---|
| input | "udru" |
| dec | "test" |

توابع CryptProtectMemory() در سیستم‌عامل ویندوز و gcry_malloc_secure() در سیستم‌عامل لینوکس، توابع معرفی‌شده توسط سیستم‌عامل برای رمزنگاری داده‌ها درون حافظه هستند.

ساختار تابع CryptProtectMemory() که در هدر wincrypt.h قرار دارد به‌صورت زیر است:

```
BOOL WINAPI CryptProtectMemory(
  _Inout_ LPVOID pData,
  _In_    DWORD  cbData,
  _In_    DWORD  dwFlags
);
```

dwFlags در این تابع نحوه رمزنگاری و رمزگشایی را تعیین می‌کند.

مقادیر dwFlags به‌صورت زیر از هدر Wincrypt.h قابل انتخاب‌اند:

| Value | Meaning |
|---|---|
| CRYPTPROTECTMEMORY_SAME_PROCESS | Encrypt and decrypt memory in the same process. An application running in a different process will not be able to decrypt the data. |
| CRYPTPROTECTMEMORY_CROSS_PROCESS | Encrypt and decrypt memory in different processes. An application running in a different process will be able to decrypt the data. |
| CRYPTPROTECTMEMORY_SAME_LOGON | Use the same logon credentials to encrypt and decrypt memory in different processes. An application running in a different process will be able to decrypt the data. However, the process must run as the same user that encrypted the data and in the same logon session. |

[۶۶]

با استفاده از تابع CryptUnprotectMemory() می‌توان اطلاعات رمزنگاری‌شده را رمزگشایی کرد.

قطعه کد زیر از این توابع برای رمزنگاری و رمزگشایی در حافظه استفاده می‌کند:

[۶۶] https://msdn.microsoft.com/en-us/library/windows/desktop/aa380262%28v=vs.85%29.aspx?f=255&MSPPError=-2147217396

```
#include <windows.h>
#include <stdio.h>
#include <wincrypt.h>

union MessageBuffer
{
	DWORD secret;
	char buffer[CRYPTPROTECTMEMORY_BLOCK_SIZE];
};
static_assert(sizeof(DWORD) <= CRYPTPROTECTMEMORY_BLOCK_SIZE, "Need a
bigger buffer");

int main(void)
{
	MessageBuffer message;
	message.secret = GetTickCount();
	printf("Shhh... the secret message is %u\n", message.secret);
	CryptProtectMemory(message.buffer, sizeof(message.buffer),
CRYPTPROTECTMEMORY_SAME_PROCESS);
	printf("You can't see it now %u\n", message.secret);
	CryptUnprotectMemory(message.buffer, sizeof(message.buffer),
CRYPTPROTECTMEMORY_SAME_PROCESS);
	printf("Was the secret message %u?\n", message.secret);
	SecureZeroMemory(message.buffer, sizeof(message.buffer));
	return 0;
}
```

خروجی قطعه کد بالا به صورت زیر است:

```
Shhh... the secret message is 575869859
You can't see it now 2046876575
Was the secret message 575869859?
```

همانگونه که مشاهده می شود، بعد از اعمال تابع CryptProtectMemory، مقدار secret به مقداری نامربوط به مقدار اولیه تبدیل شده و سپس توسط تابع CryptUnprotectMemory به مقدار اولیه باز می گردد.

حالت کلی‌تر این تابع، تابع CryptProtectData() است که داده را رمزنگاری می‌کند.

در حالت پیش‌فرض، این تابع با استفاده از اطلاعات کاربر واردشده به سیستم‌عامل ویندوز، عمل رمزنگاری را انجام می‌دهد و عمل رمزگشایی نیز باید توسط همان کاربر و بر روی همان رایانه انجام شود. این تابع برای رمزنگاری اطلاعات به‌صورت درازمدت مناسب نیست اما برای رمزنگاری اطلاعات بر روی حافظه مطلوب است.

ساختار این تابع به‌صورت زیر است:

```
BOOL WINAPI CryptProtectData(
  _In_       DATA_BLOB                 *pDataIn,
  _In_opt_   LPCWSTR                   szDataDescr,
  _In_opt_   DATA_BLOB                 *pOptionalEntropy,
  _Reserved_ PVOID                     pvReserved,
  _In_opt_   CRYPTPROTECT_PROMPTSTRUCT *pPromptStruct,
  _In_       DWORD                     dwFlags,
  _Out_      DATA_BLOB                 *pDataOut
);
```

## ۶-۶ ویرایش حافظه[۶۷]

در سیر برنامه نیاز است که اطلاعاتی همانند نام کاربری واردشده توسط کاربر درون حافظه قرارگرفته و تا زمان بسته شدن برنامه، درون حافظه باقی بماند.

دراین‌بین ممکن است شخص حمله‌کننده با استفاده از ویرایش حافظه، اطلاعات درون حافظه را تغییر داده و اطلاعات موردنظر خود را با آن جایگزین کند.

رمزنگاری اطلاعات با روش‌های گفته‌شده در بخش فاش شدن اطلاعات می‌تواند از این امر تا حدودی جلوگیری کند اما در صورت ویرایش حافظه و تغییر اطلاعات رمزنگاری‌شده به اطلاعاتی نامفهوم، ممکن است باعث رفتار تعریف‌نشده و ازکارافتادن برنامه شود.

همچنین در صورت فاش شدن نحوه رمزنگاری حافظه، امکان تغییر اطلاعات توسط شخص حمله‌کننده فراهم می‌شود.

یکی از روش‌های جلوگیری از ویرایش حافظه، استفاده از متد فقط خواندنی[۶۸] کردن بخش از حافظه است.

تابع mprotect() در سیستم‌عامل خانواده Unix و تابع VirtualProtect() در سیستم‌عامل ویندوز برای این منظور استفاده می‌شوند.

---

[۶۷] Memory Editing

[۶۸] Read-Only

مرکز مدیریت امداد و هماهنگی عملیات زیرساخت‌های داده‌ای

ساختار تابع VirtualProtect() به‌صورت زیر است:

```
BOOL WINAPI VirtualProtect(
  _In_  LPVOID lpAddress,
  _In_  SIZE_T dwSize,
  _In_  DWORD  flNewProtect,
  _Out_ PDWORD lpflOldProtect
);
```

این تابع در هد Windows.h است.

flNewProtect یک ثابت محافظ حافظه[۶۹] است.

این تابع ویژگی‌های امنیتی را به یک صفحه[۷۱] کامل تخصیص می‌دهد و امکان تخصیص این مقادیر به قسمتی از صفحه وجود ندارد.

قطعه کد زیر نحوه استفاده از این تابع را نمایش می‌دهد:

```
unsigned long oldProtect;
DWORD a;
VirtualProtect((LPVOID)a, 4, PAGE_EXECUTE_READ, &oldProtect);
```

باید دقت داشت که به‌هیچ‌وجه نباید بعد از تبدیل قسمتی از حافظه به حالت فقط خواندنی، عملیات نوشتن را بر روی آن انجام داد چراکه باعث بروز رفتار تعریف‌نشده می‌شود.

برای بازگردانی قسمت محافظت‌شده به حالت عادی و یا اعمال تغییرات بر روی آن کافی است پارامتر flNewProtect را در تابع با دسترسی موردنظر، تعویض کنیم.

لیست ورودی‌های flNewProtect که ثابت محافظ حافظه می‌باشند در پیوست ۳ آمده است.

دقت داشته باشید که فقط خواندنی شدن حافظه باید در کنار رمزنگاری حافظه قرار گیرد چراکه فقط خواندنی شدن حافظه تضمین کاملی برای جلوگیری از ویرایش حافظه نیست و به‌وسیله راهکارهایی قابل دور زدن[۷۲] است.

---

Memory Protection Constant[۶۹]

Attribute[۷۰]

Page[۷۱]

Bypass[۷۲]

## ۷-۶ سرریز بافر[۷۳]

سرریز بافر یکی از رایج‌ترین و خطرناک‌ترین آسیب‌پذیری‌های مرتبط با حافظه است.

آسیب‌پذیری سرریز بافر زمانی رخ می‌دهد که بخش از حافظه را در اختیار گرفته اما در زمان نوشتن بر روی حافظه از حدود آن بخش تجاوز می‌کنیم. در این حالت سرریز بافر رخ می‌دهد.

این آسیب‌پذیری ممکن است باعث بروز خطا، ازکارافتادن برنامه و درنهایت منجر به اجرای کد دلخواه شخص حمله‌کننده شود.

دودسته اصلی این آسیب‌پذیری سرریز پشته Stack[۷۴] و سرریز پشته Heap[۷۵] می‌باشند.

تفاوت این دودسته در مکان رخ‌دادن سرریز است.

قطعه کد زیر نمونه‌ای از سرریز Stack است:

```
#include <iostream>

void sof(char *z)
{
	char c[10];
	strcpy(c, z);
}
int main(void)
{
	int i = 0;
	std::cin >> 1;
	if (std::cin.fail())
	{
		sof("AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA");
		std::cout << "Invalid Integer!!!";
	}
	return 0;
}
```

در قطعه کد بالا، در صورت بروز خطا در ورود عدد صحیح که می‌تواند مربوط به ورود نوع داده نامعتبر باشد، تابع sof فراخوانی شده و رشته ورودی به تابع sof بیش از ۱۰ کاراکتر است و تابع strcpy بدون

---

Buffer Overflow[۷۳]

Stack Overflow[۷۴]

Heap Overflow[۷۵]

توجه به محدوده[۷۶] آرایه کاراکتری c، محتویات z را درون آن کپی می‌کند، درنتیجه باعث بروز سرریز Stack می‌شود.

خروجی قطعه کد بالا در صورت ورود یک کاراکتر حرفی به‌صورت زیر است:

```
EAX = 004FFDB8 EBX = 003B9000 ECX = 012EC218 EDX = 00000000 ESI = 012E1235
EDI = 012E1235 EIP = 41414141 ESP = 004FFDD0 EBP = 41414141 EFL = 00010206
```

مقدار ثبات EIP[۷۷] برابر با ۴۱۴۱۴۱۴۱ است که کد اسکی و معادل کاراکتر A در مبنای Hex است.

در حالتی دیگر از سرریز پشته، فضای پشته Stack به اتمام رسیده و سرریز Stack رخ می‌دهد.

قطعه کد زیر نمونه‌ای از این حالت است:

```
void sof()
{
        sof();
}

int main(void)
{
        soft();
        return 0;
}
```

با فراخوانی تابع sof() به‌صورت بازگشتی توسط خودش، فضای Stack به‌طور کامل پرشده و سرریز Stack رخ می‌دهد.

دسته بعدی سرریز Heap است. قطعه کد زیر نمونه‌ای از این آسیب‌پذیری است:

```
#include <iostream>

int main(char **argv)
{
        char *buf = (char *)std::malloc(256);
        strcpy(buf, argv[1]);
        return 0;
}
```

در قطعه کد بالا، با استفاده از malloc حافظه از Heap تخصیص‌یافته و تابع strcpy بدون رعایت حدود، اقدام به عمل کپی می‌کند.

---

[۷۶] Bound

[۷۷] Register

برای جلوگیری از بروز آسیب‌پذیری‌های سرریز بافر باید به حدود در زمان انجام عملیات و ورود و خروج بر روی حافظه دقت داشت. همچنین باید سعی کرد از توابعی که حدود را در انجام عملیات مدنظر قرار می‌دهند استفاده کرد.

یک راهکار برای جلوگیری از سرریز Stack، پیاده‌سازی Stack غیر اجرایی[۷۸] است که بسته به سیستم‌عامل و ساختار پردازنده راهکارهای متفاوتی دارد.

سیستم‌عامل و کامپایلرهای مختلف نیز راهکارهای امنیتی مختلفی برای جلوگیری از بروز این دست از آسیب‌پذیری‌ها به‌خصوص سرریز Stack ارائه می‌دهند. راهکارهای امنیتی‌ای ازجمله DEP[۸۰], ASLR[۷۹] GS, Stack Cookie(Canary) ازاین‌دست راهکارها می‌باشند.

به چند راهکار کامپایلرهای معروف می‌پردازیم:

در کامپایلر GCC[۸۱] با استفاده از -fstack-protector- از توابع آسیب‌پذیر محافظت می‌کند و -fstack-protector-all از تمامی توابع چه نیاز به محافظت داشته باشند و چه بی‌نیاز به محافظت باشند، محافظت می‌کنند. -fstack-protector-strong نسخه جدیدتر -fstack-protector است که برای تعادل هرچه بیشتر امنیت و کارایی ارائه‌شده است و از GCC 4.9 به بعد موجود است.

StackGuard و ProPolice نیز از دیگر راهکارها هستند که به دلیل عدم محافظت کامل امروزه مورداستفاده قرار نمی‌گیرند.

در Microsoft Visual Studio از نسخه ۲۰۰۳ به بعد متد امنیتی GS برای محافظت تعبیه‌شده است. با استفاده از GS/ می‌توان آن را فعال کرد و از نسخه ۲۰۰۵ به بعد به‌صورت خودکار فعال است و با استفاده از —GS/ می‌توان آن را غیرفعال کرد.

در Intel C++ Compiler، متد امنیتی Intel MPX[۸۲] تعبیه‌شده است که عملکردی همانند GS دارد.

---

Non-Executable Stack[۷۸]

Address space layout randomization[۷۹]

Data Execution Prevention[۸۰]

GNU Compiler Collection[۸۱]

Intel Memory Protection Extensions[۸۲]

در IBM Compiler با استفاده از qstackprotect- می‌توان برنامه را محافظت کرد.

در سیستم‌عامل ویندوز و کامپایلر Microsoft Visual Studio با استفاده از /Wl و /nxcompat می‌توان متد امنیتی DEP را فعال کرد. با استفاده از /dynamicbase می‌توان متد امنیتی ASLR را فعال کرد.

## ۸-۶ قالب رشته بازرسی نشده[۸۳]

آسیب‌پذیری قالب رشته بازرسی نشده اجازه کنترل خروجی توابع خانواده printf را به کاربر تأیید نشده می‌دهد. اپراتور % در این خانواده اجازه کنترل بر روی این خانواده را فراهم می‌سازند. خانواده printf در پیوست شماره ۱ به نمایش درآمده‌اند.

اپراتور %n باعث نوشته شدن شمار کاراکترهای واردشده در قسمتی از حافظه می‌شود. با استفاده از این کاراکتر شخص حمله‌کننده می‌تواند با طراحی قالب رشته‌های مختلف مقادیر موردنظر خود را در مکان موردنظر ذخیره کند.

به این حالت، وضعیت کجا چه چیزی را بنویس[۸۴] گفته می‌شود.

قطعه کد زیر نمونه‌ای از آسیب‌پذیری قالب رشته بازرسی نشده است:

```
#include <iostream>

int main(void)
{
	char *a;
	std::cin >> a;
	printf(a);
	return 0;
}
```

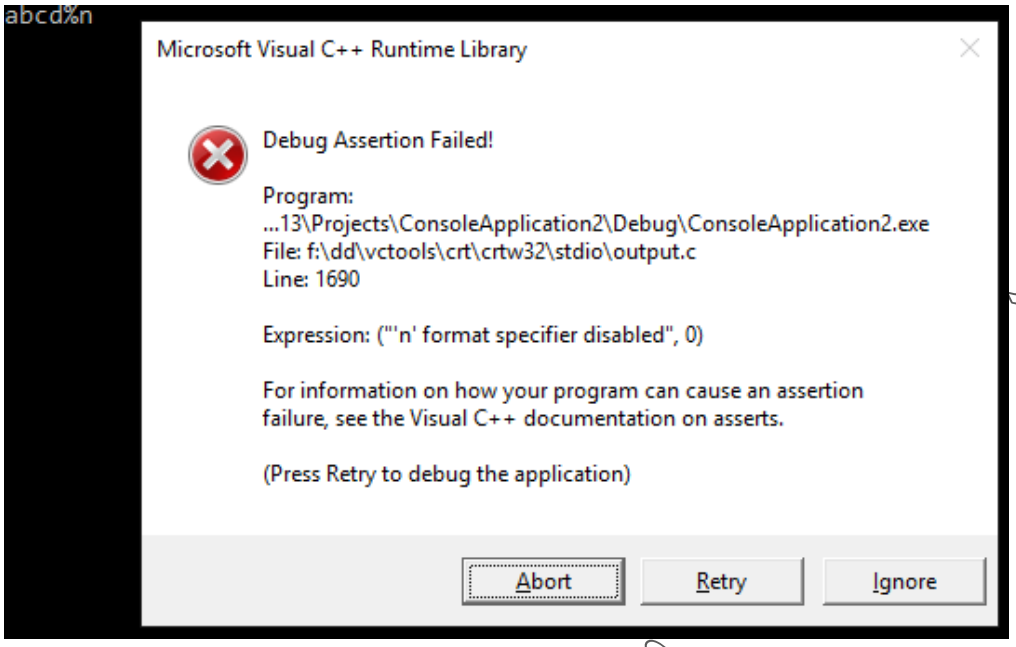
در قطعه کد بالا با استفاده از اپراتور % می‌توان کنترل تابع printf() را در دست گرفته و با استفاده از %n می‌توان از این تابع برای اجرای کد دلخواه بهره‌برداری کرد.

---

[۸۳] Uncontrolled Format String

[۸۴] Write-what-where Condition

خروجی قطعه کد بالا به‌صورت زیر است:




که به دلیل موارد امنیتی با پیغام خطا مواجه می‌شود.

برای جلوگیری از بروز این آسیب‌پذیری توصیه می‌شود تا حد امکان از %n در قالب رشته‌ها استفاده‌نشده و در صورت استفاده به‌طور کامل شرایط ایمن فراهم گردد.

ورودی‌های کاربر باید به‌طور کامل کنترل و ارزیابی شود و سپس برای نمایش درون تابع قرار گیرند.

همچنین برای نمایش کاراکترهای متنی از %s استفاده شود که به مقادیر نمایش داده‌شده ارزش رشته‌ای می‌دهد و آن‌ها را از حالت کنترلی خارج می‌سازد.

قطعه کد زیر نمونه اصلاح‌شده قطعه کد بالا است:

```
#include <iostream>

int main(void)
{
        char *a;
        std::cin >> a;
        printf("%s", a);
        return 0;
}
```

خروجی قطعه کد بالا به‌صورت زیر است:

```
abcd%n
abcd%n
```

## ۹-۶ سرریز عدد صحیح[۸۵]

سرریز عدد صحیح به‌نوعی زیرمجموعه‌ای از سرریز بافر است. سرریز عدد صحیح زمانی رخ می‌دهد که حاصل عملیات انجام‌شده مقداری باشد که از حداکثر (و یا حداقل) مقدار مجاز برای عدد صحیح تجاوز کند. در این حالت عدد صحیح درون یک چرخه قرارگرفته و از بالاترین مقدار قابل‌قبول توسط عدد صحیح به پایین‌ترین مقدار قابل‌قبول تغییر پیداکرده و عملیات ادامه پیدا می‌کند. بدیهی است که در حالتی که مقدار عدد صحیح از حداقل مقدار قابل‌قبول تجاوز کند به‌صورت برعکس و به حداکثر مقدار قابل‌قبول تغییر پیدا می‌کند و عملیات ادامه پیدا می‌کند. به این حالت زیرریز عدد صحیح[۸۶] گفته می‌شود.

به‌عنوان‌مثال اگر ساختار عدد صحیح ما یک عدد ۸ بیتی و با علامت باشد حداکثر مقدار قابل‌قبول ما ۱۲۷ و حداقل مقدار قابل‌قبول ما ۱۲۸- است. چنانچه ما به عدد ۱۲۷ یک واحد اضافه کنیم انتظار داریم که عدد ۱۲۸ جایگزین عدد فعلی شود درصورتی‌که عدد ۱۲۸- جایگزین آن خواهد شد. در صورت اضافه کردن یک واحد دیگر نتیجه ۱۲۷- می‌شود.

قطعه کد زیر نمونه‌ای از سرریز عدد صحیح است:

```
int main(void)
{
        int i = std::numeric_limits<int>::max();
        i++;
        return 0;
}
```

خروجی قطعه کد بالا به‌صورت زیر است:

| Name | Value |
|---|---|
| i | -2147483648 |

راه عمومی برای جلوگیری از سرریز و زیرریز عدد صحیح استفاده از عبارات شرطی و منطقی برای جلوگیری از بروز آن است.

---

Integer Overflow[۸۵]

Integer Underflow[۸۶]

برای بیان بهتر، از یک نمونه کاربردی‌تر استفاده می‌کنیم.

قطعه کد زیر یک تابع برای بیان نمونه‌ای از بروز سرریز و زیرریز عدد صحیح در عملیات جمع است:

```
int add(int a,int b)
{
	return a+b;
}
```

بروز آسیب‌پذیری در این کد بدیهی است.

حال برای تشخیص سرریز و زیرریز عدد صحیح، با توجه به رابطه جمع، حاصل یک جمع برابر جمع مقادیر a و b است درنتیجه مقدار a برابر کسر حاصل جمع از مقدار b است و مقدار b برابر کسر حاصل جمع از a است. ابتدا عدد علامت عدد اول ورودی که مقدار a است را مدنظر گرفته و سپس مقدار b را با تفاضل مقدار a از حداکثر و حداقل مقدار عدد صحیح مقایسه می‌کنیم تا سرریز و زیرریز قابل‌تشخیص باشد.

قطعه کد زیر نمایانگر کد ایمن شده در برابر این آسیب‌پذیری است:

```
int add(int a,int b)
{
	if (a >= 0 && b > std::numeric_limits<int>::max - a)
	{
		//Overflow
	}
	else if (a < 0 && b < std::numeric_limits<int>::max - a)
	{
		//Underflow
	}
	return a+b;
}
```

که بنا به تصمیم برنامه‌نویس اقدامات لازم در زمان بروز آسیب‌پذیری اعمال می‌شود.

راه دیگر استفاده از توابع کتابخانه‌ای و هدرهای ایمن شده در برابر این حملات است. استفاده از هدر Safeint به‌عنوان یک راهکار ایمن برای کار با اعداد صحیح توصیه می‌شود و این هدر توسط مایکروسافت مورداستفاده قرار می‌گیرد.

قطعه کد زیر استفاده از Safeint را به نمایش می گذارد:

```
#include <safeint.h>
#include <limits>

int main(void)
{
	SafeInt<int> a = std::numeric_limits<int>::max;
	SafeInt<int> b = a + 1;
	return 0;
}
```

## ۱۰-۶ خرابی حافظه[۸۷]

هر اتفاقی که باعث شود گونه‌های موجود در برنامه قرارگرفته در حافظه، به گونه‌های غیر معتبر ازلحاظ ساختاری و یا مقداری تبدیل شوند باعث خرابی حافظه می‌شود.

خرابی حافظه وقتی رخ می‌دهد که یک‌گونه ازلحاظ ساختاری و یا مقداری، متفاوت با گونه موردنظر توسط برنامه‌نویس است.

آسیب‌پذیری‌های ذکرشده در مورد حافظه و اشاره‌گر، باعث خرابی حافظه می‌شوند.

مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

[۸۷] Memory Corruption

# ۷ هم‌زمانی[۸۸]

## ۷-۱ ایمنی نخ[۸۹]ها

یکی از مباحث برنامه‌نویسی ایمن و تدافعی، هم‌زمانی نخ‌ها است. با توجه به فراگیر بودن کاربری نخ‌ها، نیاز است که در زمان برنامه‌نویسی از کارکرد صحیح آن‌ها در زمان اجرا اطمینان حاصل کنیم.

ابتدا نکات کلی در مورد ایمنی نخ‌ها گفته می‌شود و سپس حالت‌ها و آسیب‌پذیری‌های آن به‌صورت جداگانه بررسی می‌شود.

نکات کلی در ایمنی نخ‌ها:

۱- شناسایی منابع مشترک بین نخ‌ها و اطمینان از ایمنی و پردازش صحیح بر روی آن‌ها
۲- عدم استفاده از اشیاء و ساختارهای سراسری[۹۰] و پاک‌سازی آن‌ها در صورت امکان
۳- عدم استفاده از صفت static در توابع و منابع مشترک

## ۷-۲ وضعیت پیشی گرفتن[۹۱]

وضعیت پیشی گرفتن زمانی رخ می‌دهد که دو یا چند نخ[۹۲] به‌صورت هم‌زمان به یک داده مشترک دسترسی داشته و شروع به انجام تغییرات و پردازش بر روی آن کنند. به دلیل این‌که الگوریتم زمان‌بندی نخ[۹۳] ممکن است در هر زمان بین نخ‌ها تغییر کند، به‌طور دقیق نمی‌توان گفت که کدام نخ در حال حاضر در حال

---

Concurrency[۸۸]
Thread Safety[۸۹]
Global[۹۰]
Race Condition[۹۱]
Thread[۹۲]
Thread Scheduling Algorithm[۹۳]

عملیات بر روی‌داده مشترک است و نخ‌ها در حالت پیشی گرفتن برای دسترسی و تغییر درداده قرار می‌گیرند.

قطعه کد زیر نمونه‌ای از این آسیب‌پذیری است:

```
#include <iostream>
#include <thread>

void print_thread_id(int id) {
	std::cout << "thread #" << id << "\n";
}

int main(void)
{
	std::thread threads[10];
	for (int i = 0; i < 10; i++)
		threads[i] = std::thread(print_thread_id, i + 1)
	for (auto& th : threads)
		th.join();
	return 0;
}
```

در این قطعه کد ۱۰ نخ ساخته‌شده و شماره‌های آن‌ها به نمایش درمی‌آید.

خروجی قطعه کد بالا به‌صورت زیر است:

```
thread #thread #2
1
thread #3
thread #4
thread #5
thread #6
thread #7
thread #8
thread #9
thread #10
```

همان‌طور که مشاهده می‌شود در هنگام انجام عمل نمایش شماره نخ ۱، نخ ۲ از آن پیشی گرفته است.

با اجرا کردن مجدد برنامه، خروجی به‌صورت زیر است:

```
thread #1
thread #3thread #2
thread #5thread #thread #7

4

thread #6
thread #9
thread #8
thread #10
```

مرکز مدیریت امداد و هماهنگی ارتباطات و حوادث رایانه ای

که در این اجرای مجدد، پیشی گرفتن نخ‌های مختلف از یکدیگر به‌وضوح دیده می‌شود.

برای جلوگیری از این آسیب‌پذیری از دو شیوه قفل‌کردن[۹۴] و تغییر وضعیت استفاده می‌شود.

درروش قفل‌کردن، منابع و داده‌های مشترک برای نخ فراخوانند قفل‌شده تا اطمینان حاصل شود که فقط یک نخ به‌صورت هم‌زمان از آن استفاده می‌کند و این قفل شدن تا زمان از قفل درآمدن توسط نخ فراخواننده ادامه خواهد داشت.

قطعه کد زیر نمونه‌ای از کد اصلاح‌شده با این روش است:

```
#include <iostream>
#include <thread>
#include <mutex>

std::mutex mx;

void print_thread_id(int id) {
      mx.lock();
      std::cout << "thread #" << id << "\n";
      mx.unlock();
}

int main(void)
{
      std::thread threads[10];
      for (int i = 0; i < 10; i++)
            threads[i] = std::thread(print_thread_id, i + 1)
      for (auto& th : threads)
            th.join();
      return 0;
}
```

با استفاده از توابع خانواده mutex می‌توان عمل قفل‌کردن و آزادسازی را انجام داد. در این قطعه کد نخ فراخواننده مالک mutex بوده و بقیه نخ‌ها تا عدم آزادسازی mutex توسط نخ فراخواننده، اجازه مالکیت و دسترسی به آن را نخواهند داشت.

خانواده Mutex شامل انواع زیر است:

۱- std::mutex: نخ فراخواننده مالکیت را داراست و تا آزادسازی آن، نخ‌های دیگر اجازه نخواهند داشت.

---

[۹۴] Lock

۲- std::recursive_mutex: نخ فراخوانند از زمان فراخوانی موفق اولین lock و یا try_lock مالکیت را داراست و در سیر برنامه امکان فراخوانی چندین‌باره این دو تابع را دارد. آزادسازی زمانی انجام می‌شود که به تعداد لاک‌ها، unlock انجام شود.

۳- std::timed_mutex: همانند mutex است با این تفاوت که تلاش برای قفل‌کردن برای مدتی معین ادامه پیدا می‌کند.

۴- std::recursive_timed_mutex: همانند std::recursive_mutex است با این تفاوت که تلاش برای قفل‌کردن برای مدتی معین ادامه پیدا می‌کند.

۵- std::shared_mutex: این نوع دارای دو سطح دسترسی است، یکی shared که اجازه مالکیت به چند نخ را می‌دهد و دیگری exclusive که اجازه مالکیت را به یک نخ می‌دهد. از این نوع، زمانی استفاده می‌شود که مخ‌های مختلف با توجه به عدم رخ دادن وضعیت پیشی گرفتن، به‌عنوان خواننده از منابع مشترک استفاده می‌کنند و فقط یک نخ اجازه نوشتن دارد.

خروجی قطعه کد بالا به‌صورت زیر است:

```
thread #1
thread #2
thread #3
thread #4
thread #5
thread #6
thread #7
thread #8
thread #9
thread #10
```

باید دقت داشت درزمانی که mutex در حالت قفل قرار دارد، از نابودی[۹۵] آن خودداری کرد.

درروش بعدی با استفاده از تغییر وضعیت[۹۶] از بروز این آسیب‌پذیری جلوگیری می‌شود.

در این روش منابع مشترک نخ‌ها در وضعیت بلوکه قرار داشته تا زمانی که نخ یک اطلاع‌رسانی[۹۷] از نخ دیگر دریافت کرده و یا زمان ایست[۹۸] فرارسد و یا آن‌که بیداری جعلی[۹۹] رخ دهد.

---

Destroy[۹۵]

Condition Variable[۹۶]

Notification[۹۷]

قطعه کد زیر نمونه‌ای از پیاده‌سازی این روش در دو نخ اصلی برنامه و نخ ایجادشده است:

```
#include <iostream>
#include <string>
#include <thread>
#include <mutex>
#include <condition_variable>

std::mutex m;
std::condition_variable cv;
std::string data;
bool ready = false;
bool processed = false;

void worker_thread(void)
{
	std::unique_lock<std::mutex> lk(m);
	cv.wait(lk, []{return ready; });
	std::cout << "Worker thread is processing data\n";
	data += " after processing";
	processed = true;
	std::cout << "Worker thread signals data processing completed\n";
	lk.unlock();
	cv.notify one();
}

int main(void)
{
	std::thread worker(worker_thread);
	data = "Example data";
	{
		stdulock_guard<stdumutex> lk(m);
		ready = true;
		std::cout << "main() signals data ready for processing\n";
	}
	cv.notify one();
	{
		stduunique_lock<stdumutex> lk(m);
		cv.wait(lk, []{return processed; 1);
	}
	std::cout << "Back in main()) data = " << data << '\n';
	worker.join();
}
```

std::condition_variable برای تعریف متغیر وضعیت به کار می‌رود.

---

Timeout[۹۸]

Spurious Wake-Up[۹۹]

در قطعه کد بالا ابتدا نخ اصلی که برای اجرای برنامه به کار می‌رود اجراشده و سپس نخ worker_thread فراخوانی می‌شود و بعد از اتمام آن عملیات چاپ متن ادامه پیدا می‌کند. درواقع نخ اصلی منتظر رسیدن یک علامت از نخ worker_thread می‌ماند.

خروجی قطعه کد بالا به‌صورت زیر است:

```
main() signals data ready for processing
Worker thread is processing data
Worker thread signals data processing completed
Back in main(), data = Example data after processing
```

مشکل محیاکننده-مصرف‌کننده[100] که به‌عنوان یک مثال کلاسیک از مشکل نخ‌ها مطرح می‌شود، درواقع نوعی از وضعیت پیشی گرفتن است.

اشیاء ساخته‌شده از انواع atomic تنها ساختارهایی هستند که از سبقت‌گیری داده[101] ایمن هستند.

توصیه می‌شود در زمان استفاده از نخ‌ها برای ایمنی بیشتر، از ساختارهای atomic استفاده شود.

انواع atomic در پیوست ۲ به نمایش درآمده‌اند.

## ۳-۷ بن‌بست[102]

بن‌بست به زمانی گفته می‌شود که یک یا چند نخ منتظر دیگری هستند تا به کار خود ادامه دهند اما هیچ‌کدام از آن‌ها شرایط ادامه کار دیگری را فراهم نمی‌آورد. این شرایط در زبان C++ عمدتاً به دلیل استفاده نادرست از mutex به وجود می‌آید.

قطعه کد زیر نمونه‌ای از بن‌بست است:

مرکز مدیریت امداد و هماهنگی عملیات واحدهای رایانه ای

[100] Producer-Consumer Problem

[101] Data Race

[102] Deadlock

```
#include <iostream>
#include <thread>
#include <mutex>

std::mutex mutex1, mutex2;

void ThreadA(void)
{
	mutex2.lock();
	std::cout << "Thread A" << std::endl;
	mutex1.lock();
	mutex2.unlock();
	mutex1.unlock();
}

void ThreadB(void)
{
	mutex1.lock();
	std::cout << "Thread B" << std::endl;
	mutex2.lock();
	mutex1.unlock();
	mutex2.unlock();
}

int main(void)
{
	std::thread t1(Thread A);
	std::thread t2(Thread B);
	t1.join();
	t2.join();
	std::cout << "Finished" << std::endl;
	return 0;
}
```

در قطعه کد بالا، mutex1 توسط ThreadB و mutex2 توسط ThreadA قفل‌شده‌اند. در این وضعیت ThreadA سعی در قفل‌کردن mutex1 و ThreadB سعی در قفل‌کردن mutex2 دارد. در این حالت بن‌بست رخ‌داده و هر نخ منتظر دیگری است.

خروجی قطعه کد بالا به‌صورت زیر است:

```
Thread AThread B
```

در این مورد، برای برطرف سازی بن‌بست می‌توان در هر دو تابع ThreadA و ThreadB از یک ترتیب قفل‌کردن mutex استفاده کرد.

مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

قطعه کد زیر نمونه اصلاح‌شده قطعه کد بالا است:

```
#include <iostream>
#include <thread>
#include <mutex>

std::mutex mutex1, mutex2;

void ThreadA(void)
{
	mutex1.lock();
	std::cout << "Thread A" << std::endl;
	mutex2.lock();
	mutex1.unlock();
	mutex2.unlock();
}

void ThreadB(void)
{
	mutex1.lock();
	std::cout << "Thread B" << std::endl;
	mutex2.lock();
	mutex1.unlock();
	mutex2.unlock();
}

int main(void)
{
	std::thread t1(Thread A);
	std::thread t2(Thread B);
	t1.join();
	t2.join();
	std::cout << "Finished" << std::endl;
	return 0;
}
```

در این حالت زمانی که mutex1 توسط ThreadA قفل می‌شود ThreadB منتظر آزادسازی توسط آن مانده و سپس به اجرا درمی‌آید و mutex2 نیز بعد از چاپ عبارت موردنظر در هر تابع قفل و آزاد می‌شود.

خروجی قطعه کد بالا به‌صورت زیر است:

```
Thread A
Thread B
Finished
```

مرکز مدیریت امداد و هماهنگی عملیات زیرساخت‌های رایانه ای

# ۸ مدیریت خطا[۱۰۳]

## ۸-۱ خطاهای منطقی[۱۰۴]

خطاهای منطقی آن دست از خطاها هستند که ازلحاظ دستوری مشکلی نداشته اما در عمل باعث ایجاد مشکل و عدم کارکرد صحیح برنامه می‌شوند.

برخی از خطاهای منطقی باعث بروز پیغام خطا در برنامه می‌شوند اما خطاهای منطقی عمدتاً باعث تغییر در سیر برنامه و رفتار تعریف‌نشده می‌شوند.

خطاهای منطقی به هر شکلی می‌توانند ایجاد شوند و منشأ بروز آن‌ها اشتباهات برنامه‌نویسی برنامه‌نویس است لذا در برخی از موارد پیدا کردن آن‌ها بسیار دشوار و زمان‌بر است بنابراین برنامه‌نویس در زمان برنامه‌نویسی باید نهایت دقت را به کار برد.

خطاهای منطقی رایج:

۱- تقسیم عدد بر ۰[۱۰۵]

۲- استفاده از اپراتورهای بیتی[۱۰۶] به‌جای اپراتورهای منطقی در عبارات شرطی (| به‌جای ||، &به‌جای && و ^ به‌جای ^^)

۳- استفاده از اپراتور تخصیص به‌جای اپراتور تساوی در عبارات شرطی (= به‌جای ==)

۴- عدم توجه به تقدم اپراتورهای محاسباتی

۵- عدم توجه به تقدم و باز و بسته کردن پرانتزها

۶- عدم توجه به عبارات محاسباتی پیشوندی[۱۰۷] و پسوندی[۱۰۸] (-- و ++)

---

Error Handling[۱۰۳]

Logical Errors[۱۰۴]

Division By Zero[۱۰۵]

Bitwise[۱۰۶]

Prefix[۱۰۷]

Postfix[۱۰۸]

مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

۷- شرط شمارش ناصحیح و اشتباه در حلقه‌ها

۸- شرط همیشه صادق و یا همیشه ناصادق حلقه‌ها

۹- نقطه‌ویرگول[۱۰۹] نابجا (به‌عنوان‌مثال آخر دستور for) (این مورد به‌عنوان خطای نوشتاری[۱۱۰] نیز مطرح می‌شود)

۱۰- عدم توجه به علامت در تبدیلات عدد صحیح با علامت به بدون علامت

۱۱- عدم توجه به بازه مجاز برای قسمت اعشار اعداد اعشاری

۱۲- عدم توجه به تبدیلات عدد اعشاری

۱۳- عدم توجه به بیت‌های عددی، بیت‌های ساختاری و بیت علامت در انجام عملیات بر روی اعداد صحیح

۱۴- عدم توجه به بازه‌های قابل‌قبول توسط اعداد اعشاری و اعداد صحیح در زمان تبدیل (که در این مورد آسیب‌پذیری‌ای همانند سرریز عدد صحیح رخ می‌دهد)

## ۸-۲ خطاهای زمان اجرا[۱۱۱]

خطاهای زمان اجرا، خطاهایی هستند که در زمان تألیف[۱۱۲] برنامه رخ نداده اما در سیر برنامه به وجود می‌آیند. البته برخی از خطاهای منطقی همانند تقسیم عدد بر صفر نیز خود را با خطاهای زمان اجرا نمایش می‌دهند.

یکی از مباحث اصلی برنامه‌نویسی، نوشتن کد ایمن در برابر خطا[۱۱۳] است. بدین معنی که تا حد امکان کد به صورتی نوشته شود که از بروز خطا در زمان اجرای آن جلوگیری شود، در مرحله بعد خطا باعث ازکارافتادن

---

Semicolon[۱۰۹]

Syntax Error[۱۱۰]

Runtime Errors[۱۱۱]

Compile[۱۱۲]

Exception-Safe Code[۱۱۳]

مرکز مدیریت امداد و هماهنگی عملیات زیرساخت‌های رایانه ای

برنامه نشود و درنهایت خطا باعث نشر اطلاعات از برنامه نشود و با اطلاعات قابل‌فهم توسط کاربر، نمایش داده شود.

قطعه کد زیر نمونه‌ای از بروز خطا در زمان اجرا است:

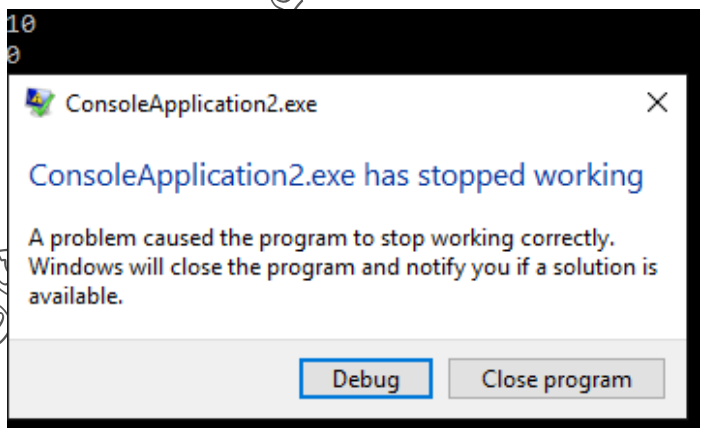
```
#include <iostream>

int main(void)
{
	int i = 0, j = 0;
	std::cin >> i >> j;
	std::cout << i / j;
	return 0;
}
```

در قطعه کد بالا، در صورت ورود صفر به‌عنوان مقدار j و یا ورود عبارت رشته‌ای به‌جای عدد صحیح به‌عنوان مقدار j، خطای منطقی تقسیم بر صفر به‌صورت خطای زمان اجرا به نمایش درمی‌آید.

بروز این خطا باعث ازکارافتادن برنامه و نمایش پیغام خطا می‌شود.

خروجی برنامه بعد از ورود عدد ۰ به‌عنوان مقدار j به‌صورت زیر است:




همان‌طور که مشخص است، یک خطای تقسیم عدد بر ۰ می‌تواند به‌طور کامل سیر اجرای برنامه را مختل کند.

برای مدیریت این خطا بر طبق اصول کد ایمن در برابر خطا، جلوگیری از بروز این خطا را انجام می‌دهیم و سپس نحوه مدیریت خطاها را به‌صورت کلی بیان می‌کنیم.

قطعه کد زیر، نحوه مدیریت این خطا را به ساده‌ترین شکل توضیح می‌دهد:

مرکز مدیریت امداد و هماهنگی های رایانه ای

```
#include <iostream>

int main(void)
{
	int i = 0, j = 0;
	std::cin >> i >> j;
	if (j == 0)
		std::cout << "Input Error...";
	else
		std::cout << i / j;
	return 0;
}
```

با استفاده از یک عبارت شرطی مقدار j بررسی‌شده و در صورت ۰ نبودن، عمل تقسیم انجام می‌شود.

تمامی خطاها بدین‌صورت قابل مدیریت نیستند.

فرض کنید برنامه نوشته‌شده قصد نوشتن بر روی پرونده متنی‌ای را دارد درحالی‌که این پرونده توسط یک برنامه دیگر بازشده و در حال پردازش است. در این زمان برنامه با خطا مواجه می‌شود.

در این زمان با استفاده از ساختار try catch، از بروز خطاهای احتمالی جلوگیری می‌شود.

قطعه کد زیر نحوه مدیریت خطای تقسیم عدد بر صفر با استفاده از این ساختار است:

```
#include <iostream>

int main(void)
{
	int i = 0, j = 0;
	std::cin >> i >> j;
	try
	{
		std::cout << i / j;
	}
	catch (...)
	{
		std::cout << "Input Error...";
	}
	return 0;
}
```

در قطعه کد بالا، در صورت ورود عدد صفر به‌عنوان مقدار j، خروجی به‌صورت زیر است:

```
10
0
Input Error...
```

همان‌طور که مشاهده می‌شود دستورات درون catch اجرا می‌شوند.

سه‌نقطه درون پرانتزهای catch به معنی دریافت هرگونه خطا است. می‌توان چندین catch پشت سر هم با شرایط مختلف داشت.

هدر exception شامل ابزارهای مدیریت خطاها و باقابلیت تشخیص انواع خطا است و می‌توان از ساختارهای موجود در آن به‌جای سه‌نقطه درون پرانتز استفاده کرد.

چنانچه در سیر برنامه نیاز باشد که خطایی به‌عمد ایجاد شود، با استفاده از دستور throw این کار انجام می‌گیرد. بعد از throw می‌توانید نوع خطا را مشخص کنید.

ورودی catch همچنین می‌تواند یکی از انواع داده‌های بنیادی و یا مشتقات آن باشد.

قطعه کد زیر نمونه‌ای از این موضوع است:

```
#include <iostream>

int main(void)
{
	int i = 0, j = 0;
	std::cin >> i >> j;
	try
	{
		if (j == 0)
			throw "Input Error...";
		std::cout << i / j;
	}
	catch (char* e)
	{
		std::cout << e;
	}
	return 0;
}
```

در این قطعه کد در صورت بروز خطا، پیغام ورودی از throw به نمایش درمی‌آید و این موضوع امکان وجود یک catch برای نمایش پیغام‌های مختلف مربوط به هر خطا را فراهم می‌سازد.

با استفاده از noexcept می‌توان از بروز خطاها جلوگیری کرد. با فعال‌سازی noexcept بر روی یک بخش، در صورت بروز خطا در آن بخش، برنامه از اعلام آن خطا چشم‌پوشی می‌کند.

std::exception به‌منظور مدیریت جامع خطا به کار می‌رود و قابل بازنویسی است.

قطعه کد زیر نمونه‌ای از چند نکته بالا است:

```
#include <exception>
#include <iostream>

struct S : std::exception {
	const char *what() const noexcept override {
		return "My custom exception";
	}
};

void f() {
	try {
		throw S();
	}
	catch (std::exception &E) {
		std::cout << E.what() << std::endl;
	}
}
```

باید دقت داشت که خطاها باید با استفاده از مقادیر ارجاعی مدیریت شوند.

در قطعه کد بالا از &E برای این منظور استفاده‌شده و استفاده از E غلط است.

مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

# ۹ مدیریت ارتباطات برون برنامه‌ای:

## ۹-۱ ارتباط با پایگاه داده

در ارتباط با پایگاه داده چند نکته قابل‌توجه است.

نخست برقراری ارتباط با پایگاه داده است که ارتباط باید از نوع ایمن بوده و به‌طور کامل رمزنگاری شود.

یکی از رایج‌ترین راه‌های ارتباط ایمن با بانک اطلاعاتی استفاده از پروتکل SSL برای برقراری ارتباط با بانک اطلاعاتی است که امری مرسوم است. همچنین سرویس‌های مختلف بانک اطلاعاتی راهکارهای امنیتی متفاوتی را برای برقراری ارتباط در اختیار می‌گذارند که استفاده از آن‌ها در برنامه‌نویسی ایمن و تدافعی ضروری است و برنامه‌نویس با توجه به کاربری برنامه و سطح امنیت آن، یک یا چند مورد امنیتی را انتخاب کرده و در برنامه خود به کار می‌گیرد.

در قدم بعد کد دستوری ارسالی به بانک اطلاعاتی برای اجرا باید از هرگونه دست‌کاری و تزریق در امان باشد. آسیب‌پذیری تزریق کد[۱۱۴] باعث می‌شود که کد دستوری ارسالی به بانک اطلاعاتی دست‌کاری شده و باعث تغییراتی در بانک اطلاعاتی و یا نتیجه بازگرداننده باشد. با توجه به جامعیت بانک‌های اطلاعاتی مبتنی بر SQL، آسیب‌پذیری تزریق SQL[۱۱۵] از رایج‌ترین دسته آسیب‌پذیری‌های تزریق کد به بانک اطلاعاتی است.

تزریق کد به بانک اطلاعاتی به دلیل فراگیری بیشتر در برنامه‌های تحت وب[۱۱۶] گاهی بر روی برنامه‌نویسی برای سیستم‌عامل نادیده گرفته می‌شوند که این امر اشتباه بوده و یک ریسک امنیتی محسوب می‌شود و جلوگیری از تزریق کد به بانک اطلاعاتی از اهمیت خاصی در برنامه‌نویسی ایمن و تدافعی برخوردار است.

قطعه کد زیر یک نمونه از اجرای دستورات SQL است:

```
std::string query = "SELET 8 FROM C_T WHERE NAME = '" + input + ";";
```

---

Code Injection[۱۱۴]

SQL Injection[۱۱۵]

Web Applications[۱۱۶]

در این قطعه کد، input توسط کاربر واردشده و به‌طور مستقیم درون کد ارسالی به بانک اطلاعاتی قرار می‌گیرد.

کد ارسالی به بانک اطلاعاتی در زمان ورود test توسط کاربر به‌صورت زیر است:

```
SELECT * FROM C_T WHERE NAME = 'test';
```

حال فرض کنید ورودی کاربر به‌صورت زیر باشد:

```
a'; UPDATE C_T SET password='123' WHERE NAME='test2
```

با قرارگیری این ورودی به‌جای input، کد ارسالی به بانک اطلاعاتی به‌صورت زیر است:

```
SELECT * FROM C_T WHERE NAME = 'a';
UPDATE C_T SET password='123' WHERE NAME='test2';
```

که نتیجه آن تغییر فیلد password مربوط به سطری که فیلد NAME آن برابر test2 است.

برای جلوگیری از این آسیب‌پذیری، باید ورودی کاربران را به‌دقت بررسی کرد، همچنین تمامی کاراکترهای غیر موردنیاز را در فیلدهای ورودی توسط کاربر بلوکه کرد و به‌هیچ‌وجه ورودی کاراکتری و رشته‌ای را به‌صورت مستقیم درون کد ارسالی به بانک اطلاعاتی قرار نگیرد.

قطعه کد نحوه جلوگیری از بروز این آسیب‌پذیری را با استفاده از لیست سفید[117] به نمایش می‌گذارد:

```
std::string::size_type off;
std::string input = "";
std::cin >> input;
off =
input.find_first_not_of("abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUV
WXYZ1234567890");
if (off != std::string::npos)
      std::cout << "Invalid Input...";
```

در قطعه کد بالا، کاراکترهای حرفی و عددی موردقبول قرار می‌گیرد و در صورت ورود کاراکتری جز کاراکترهای تعریف‌شده، پیغام موجود در عبارت شرطی به نمایش درمی‌آید.

در قطعه کد زیر، کارکتری خاص از رشته query حذف می شود:

```
query.erase(std::remove(query.begin(), query.end(), '\''), query.end());
```

Whitelist[117]

در روشی دیگر، کارکتر های خاص که باعث تغییر در عبارات دستوری SQL می شوند از رشته ورودی حذف می گردند. به این روش رهایی از کارکتر خاص[118] می گویند.

اطلاعات بااهمیت باید درون بانک اطلاعاتی به‌صورت رمزنگاری‌شده قرار گیرند و رمزنگاری مواردی همانند کلمه عبور[119] باید به‌صورت رمزنگاری بازگشت‌ناپذیر (یک‌طرفه) باشند.

وجود رمزنگاری بر روی‌داده‌های حساس باعث می‌شود که در صورت نفوذ به بانک اطلاعاتی از روش‌های دیگر، اطلاعات از دسترسی شخص نفوذکننده محفوظ بمانند.

قطعه کد زیر نحوه رمزنگاری یک‌طرفه یک‌رشته را به نمایش می‌گذارد:

```
#include <cryptopp/sha.h>
#include <cryptopp/filters.h>
#include <cryptopp/hex.h>
#include <string>

std::string generateHash(std::string source)
{
	CryptoPP:SHA512 hash;
	byte digest[CryptoPP:SHA512::DIGESTSIZE];
	hash CalculateDigest(digest, const byte*)source.c_str(),
source.size());
	std::string output;
	CryptoPP::HexEncoder encoder;
	CryptoPP::StringSink test = CryptoPP::StringSink(output);
	encoder.Attach(new CryptoPP::StringSink(output));
	encoder.Put(digest, sizeof(digest));
	encoder.MessageEnd();
	return output;
}

int main(void)
{
	std::string a = "test";
	std::string o = generateHash(a);
	return 0;
}
```

در این قطعه کد با استفاده از کتابخانه Crypto++ و متد هش SHA512 اطلاعات رمزنگاری می‌شوند.

حال با استفاده از تابع generateHash به‌راحتی می‌توان کلمه عبور رمزنگاری‌شده را در بانک اطلاعاتی ذخیره کرد و یا صحت آن را تأیید کرد.

---

Special Character Escaping[118]

Password[119]

قطعه کد زیرنمایان گر این موضوع است:

```
std::string::size_type username = "";
std::string password = "";
std::string query = "";
std::string::size_type off;
std::cin >> username >> password;
off =
username.find_first_not_of("abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRS
TUVWXYZ1234567890");
if (off != std::string::npos)
{
      query = "SELECT * FROM tbl WHERE username='" + username + "' AND
password='" + generateHash(password) + "'";
      execute(query);
}
```

در قطعه کد بالا، در صورت ورود صحیح نام کاربری و کلمه عبور، اطلاعات مربوط به کاربر بازگردانی می‌شود.

## ۹-۲ اجرای برنامه‌های بیرونی[۱۲۰]

یکی از توابع مورداستفاده برای اجرای برنامه‌های بیرونی که در زبان C معرفی شد و در زبان C++ نیز توسط برنامه‌نویسان مورداستفاده قرار می‌گیرد تابع system() است اما در برنامه‌نویسی ایمن تا حد امکان نباید از این تابع استفاده شود.

استفاده از این تابع برنامه فراخوانی شده را با سطح دسترسی برنامه فراخواننده اجرا می‌کند. همچنین برنامه فراخواننده را در وضعیت توقف قرار می‌دهد تا برنامه فراخوانده شده وضعیت خروج خود را به فراخواننده بازگرداند. استفاده از این تابع امکان اجرای برنامه‌ای جز برنامه مدنظر برنامه‌نویس را به شخص حمله‌کننده می‌دهد. رشته دستوری ورودی به آن باید به‌دقت موردبررسی قرار گیرد و محل قرارگیری برنامه و آدرس محل فراخوانی برنامه غیر قابل‌تغییر توسط اشخاص و پردازش[۱۲۱]های تأیید نشده باشد.

استفاده از این تابع همچنین باعث اخطار دهی برخی از دستگاه‌های امنیتی می‌شود.

---

[۱۲۰] External Programs

[۱۲۱] Process

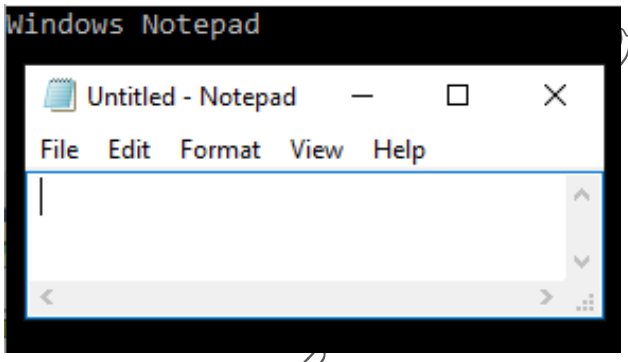
با توجه به این نکات تا حد امکان نباید از تابع system() استفاده کرد.

قطعه کد زیر نمونه‌ای از آسیب‌پذیری تابع system() در سیستم‌عامل ویندوز است:

```
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
      printf("Windows Notepad");
      system("notepad");
      return 0;
}
```

خروجی قطعه کد بالا به‌صورت زیر است:




حال برنامه‌ای به قطعه کد زیر ساخته و بانام notepad.exe در کنار برنامه نوشته‌شده فعلی قرار می‌دهیم:

```
#include <stdio.h>
#include <iostream>

int main(void)
{
      printf(" Untrusted Code");
      std::getchar();
      return 0;
}
```

بعد از اجرای کد اولیه، خروجی به‌صورت زیر است:

```
Windows Notepad Untrusted Code
```

کد موردنظر ما به‌جای برنامه notepad ویندوز اجرا می‌شود.

نمونه بالا با دادن آدرس کامل Notepad به‌جای واژه notepad درون ورودی system نیز اصلاح می‌شود اما باید دقت داشت که در صورت دست‌کاری پرونده notepad.exe توسط شخص حمله‌کننده و

یا انتقال مسیر فراخواننده به پرونده مخرب، با توجه به سطح دسترسی تابع system()، امکان اجرای کد مخرب به‌راحتی امکان‌پذیر می‌شود.

همان‌گونه که گفته شد، بعد از استفاده از تابع system()، برنامه در انتظار بازگشت وضعیت خروج از برنامه فراخواننده می‌ماند.

فرض کنید برنامه فراخوانده شده، قطعه کد زیر باشد:

```
int main(void)
{
	while (1)
	{
	}
	return 0;
}
```

این قطعه کد هیچ‌وقت مقدار بازگشتی وضعیت خروج نداشته و باعث ازکارافتادن برنامه اصلی می‌شود.

در قدم بعد نیاز به بررسی سطح دسترسی این تابع است. برای بیان و فهم راحت‌تر سطح دسترسی تابع system() به قطعه کد زیر توجه کنید:

```
#include <string.h>
#include <stdlib.h>
#include <iostream>

enum { BUFFERSIZE = 512}

void func(const char *input) {
	char cmdbuf[BUFFERSIZE];
	int len_wanted = sprintf_s(cmdbuf, BUFFERSIZE,
	"any_cmd '%s'", input);
	if (len_wanted >= BUFFERSIZE) {
		/* Handle error */
	}
	else if (len_wanted < 0) {
		/* Handle error */
	}
	else if (system(cmdbuf) == -1) {
		/* Handle error */
	}
}
```

در این قطعه کد از تابع system() برای اجرای any_cmd استفاده‌شده است.

حال فرض کنید ورودی تابع در سیستم‌عامل ویندوز مقدار زیر باشد:

```
test & shutdown -s
```

در این حالت تزریق کد صورت گرفته و نتیجه اجرای این دستور با توجه به سطح دسترسی تابع سیستم می‌تواند خاموش شدن کامل رایانه باشد.

راه‌حل ایمن استفاده از تابع CreateProcess() در ویندوز و توابع خانواده exec در لینوکس است.

ساختار تابع CreateProcess() به‌صورت زیر است:

```
BOOL WINAPI CreateProcess(
  _In_opt_    LPCTSTR               lpApplicationName,
  _Inout_opt_ LPTSTR                lpCommandLine,
  _In_opt_    LPSECURITY_ATTRIBUTES lpProcessAttributes,
  _In_opt_    LPSECURITY_ATTRIBUTES lpThreadAttributes,
  _In_        BOOL                  bInheritHandles,
  _In_        DWORD                 dwCreationFlags,
  _In_opt_    LPVOID                lpEnvironment,
  _In_opt_    LPCTSTR               lpCurrentDirectory,
  _In_        LPSTARTUPINFO         lpStartupInfo,
  _Out_       LPPROCESS_INFORMATION lpProcessInformation
);
```

باید دقت داشت که از ارسال مقدار NULL به‌عنوان lpApplicationName خودداری کرد و در صورت ارسال NULL، از علامت نقل‌قول[۱۲۲] در ابتدا و انتهای مسیر اجرایی در lpCommandLine استفاده کرد.

قطعه کد زیر نمونه‌ای از استفاده از این تابع برای اجرای test.exe است:

```
#include <Windows.h>

void func(TCHAR *input) {
      STARTUPINFO si = { 0 };
      PROCESS_INFORMATION pi;
      si.cb sizeof(si);
      CreateProcess(TEXT("test.exe"), input, NULL, NULL, FALSE,
            0, 0, 0, &si, &pi);
            CloseHandle(pi.hThread);
            CloseHandle(pi.hProcess);
}
```

نتیجه اجرای این تابع، ایجاد یک پردازش فرزند[۱۲۳] است که هیچ کنترل‌کننده‌ای را از پدر خود به ارث نمی‌برد.

---

Quotation Mark[۱۲۲]

باید دقت داشت که مقدار input نباید ایستا باشد چراکه تابع CreateProcess() در زمان فراخوانی آن را ویرایش می‌کند.

## ۳-۹ پرونده‌های موقت[۱۲۴]

پرونده‌های موقت پرونده‌هایی هستند که در زمان اجرای برنامه ایجادشده و در سیر برنامه و یا بعد از اتمام کار برنامه بی‌مصرف می‌شوند. این پرونده‌ها کمک می‌کنند تا در مصرف حافظه اصلی صرف جویی شود و داده‌های حجیمی که فضایی برای آن‌ها در حافظه اصلی نیست را نگه‌داری می‌کنند.

فرض کنید برنامه‌ای برای یک روز به جمع‌آوری داده از طریق شبکه می‌پردازد و سپس بعد از اتمام این زمان بر روی این داده‌ها پردازش انجام می‌دهد. بدیهی است که این حجم از داده را نباید درون حافظه اصلی ذخیره کرد چراکه ممکن است برنامه با مشکل کمبود حافظه مواجه شود. پرونده‌های موقت بهترین راهکار برای این سناریو می‌باشند.

ایمنی در ساخت و استفاده از پرونده‌های موقت باید رعایت شود چراکه در صورت عدم رعایت ایمنی، ممکن است پرونده‌های موقت دست‌کاری شده و یا دسترسی به آن‌ها غیرممکن شود که نتیجه آن ازکارافتادن برنامه و یا رفتار تعریف‌نشده و سوءاستفاده شخص حمله‌کننده است.

تا حد امکان باید از ساخت پرونده‌های موقت در مسیرهای اشتراکی %temp% در ویندوز و /temp و /var/temp در لینوکس جلوگیری کرد.

در صورت ساخت پرونده موقت، رعایت نکات زیر ضروری است:

۱- نام پرونده باید غیرقابل حدس زدن باشد.
۲- نام پرونده باید منحصربه‌فرد باشد.
۳- در صورت موجود نبودن به‌صورت atomic ساخته و باز شود.
۴- با دسترسی انحصاری ایجاد و باز شود.

---

[۱۲۳] Child Process
[۱۲۴] Temporary Files

۵- با سطح دسترسی مناسب ایجاد و باز شود.

۶- قبل از خروج از برنامه، پاک شود.

تابع mkstemp() در سیستم‌عامل لینوکس برای ایمن‌سازی پرونده‌های موقت به کار می‌رود.

تابع LockFile() در سیستم‌عامل ویندوز، ناحیه پرونده را قابل‌خواندن توسط تمامی پردازش‌ها می‌کند اما عملیات نوشتن در انحصار پردازشی است که این تابع را فراخوانی کرده است.

تابع LockFileEx() در سیستم‌عامل ویندوز، ناحیه پرونده را با سطح دسترسی نامحدود در اختیار پردازش فراخواننده قرار داده و تمامی دسترسی‌ها را برای پردازش‌های دیگر ممنوع می‌کند.

ساختار این توابع به‌صورت زیر است:

```
BOOL WINAPI LockFile(
  _In_ HANDLE hFile,
  _In_ DWORD  dwFileOffsetLow,
  _In_ DWORD  dwFileOffsetHigh,
  _In_ DWORD  nNumberOfBytesToLockLow,
  _In_ DWORD  nNumberOfBytesToLockHigh
);
```

```
BOOL WINAPI LockFileEx(
  _In_        HANDLE       hFile,
  _In_        DWORD        dwFlags,
  _Reserved_  DWORD        dwReserved,
  _In_        DWORD        nNumberOfBytesToLockLow,
  _In_        DWORD        nNumberOfBytesToLockHigh,
  _Inout_     LPOVERLAPPED lpOverlapped
);
```

hFile یک رسیدگی‌کننده به پرونده است که باید با سطح دسترسی GENERIC_READ و یا GENERIC_WRITE ساخته‌شده باشد.

dwFlags در تابع LockFileEx() بیان‌گر نحوه قفل‌سازی است که با استفاده از مقادیر زیر، مقداردهی می‌شود:

| Value | Meaning |
|---|---|
| **LOCKFILE_EXCLUSIVE_LOCK** 0x00000002 | The function requests an exclusive lock. Otherwise, it requests a shared lock. |
| **LOCKFILE_FAIL_IMMEDIATELY** 0x00000001 | The function returns immediately if it is unable to acquire the requested lock. Otherwise, it waits. |

از توابع UnlockFile() و UnlockFileEX() نیز برای از قفل درآوردن استفاده می‌شوند.

قطعه کد زیر نمونه‌ای از ساخت پرونده موقت ایمن با استفاده از این توابع است:

```
#define NUMWRITES 10
#define TESTSTRLEN 11

const char TestData[NUMWRITES][TESTSTRLEN] =
{
      "TestData0\n",
      "TestData1\n",
      "TestData2\n",
      "TestData3\n",
      "TestData4\n",
      "TestData5\n",
      "TestData6\n",
      "TestData7\n",
      "TestData8\n",
      "TestData9\n",
};

HANDLE hFile = CreateFile(TEXT("datafile.txt"),
      GENERIC_READ | GENERIC_WRITE,
      0,
      NULL,
      CREATE_NEW,
      0,
      NULL);
DWORD dwNumBytesWritten = 0;
for (int i = 0; i < NUMWRITES; i++)
{
      WriteFile(hFile,
            TestData[i],
            TESTSTRLEN,
            &dwNumBytesWritten,
            NULL);
}
FlushFileBuffers(hFile);
OVERLAPPED sOverLapped;
sOverLapped.Offset = TESTSTRLEN * 3;
sOverLapped.OffsetHigh = 0;
LockFileEx(hFile,
      LOCKFILE_EXCLUSIVE_LOCK |
      LOCKFILE_FAIL_IMMEDIATELY,
      0,
      TESTSTRLEN,
      0,
      &sOverLapped);
```

در قطعه کد بالا ابتدا پرونده datafile.txt ساخته‌شده و سپس با استفاده از LockFileEx() در اختیار پردازش فعلی قرار می‌گیرد.

```
UnlockFileEx(hFile,
      0,
      TESTSTRLEN,
      0,
      &sOverlapped);
```

مرکز مدیریت امداد و هماهنگی سیلاب

```
CloseHandle(hFile);
DeleteFile(TEXT("datafile.txt"));
```

سپس پرونده از قفل درآمده و پاک می‌شود.

باید دقت داشت که قبل از پاک‌سازی حتماً باید پرونده را از حالت قفل درآورد چراکه در غیر این صورت حالت تعریف‌نشده رخ می‌دهد.

## ۹-۴ زمان بازرسی، زمان استفاده[۱۲۵]

زمان بازرسی–زمان استفاده در واقع یک نوع حالت پیشی گرفتن است که در آن چند پردازش به‌طور هم‌زمان به یک پرونده مشترک دسترسی دارند.

در این حالت، برنامه‌نویس در دسترسی اول برای اطمینان ابتدا پرونده موردنظر را ازنظر ویژگی‌ها و دسترسی بازرسی کرده سپس در دسترسی بعدی شروع به انجام عملیات می‌کند. دراین‌بین شخص حمله‌کننده می‌تواند در زمان اجرایی بین این دو دسترسی پرونده را تغییر دهد و یا مسیر پرونده را به پرونده ثانویه‌ای اشاره دهد.

این آسیب‌پذیری معمولاً زمانی که برنامه دو یا چند عملیات را بر روی یک پرونده انجام می‌دهد رخ می‌دهد و این حالت باعث به وجود آمدن دریچه پیشی گرفتن[۱۲۶] می‌شود.

قطعه کد زیر نمونه‌ای از این حالت است:

```
void open_some_file(const char *file) {
      FILE *f = fopen(file, "r");
      if (NULL != f) {
            /* FILE exists, handle error */
      }
      else {
            if (fclose(f) == EOF) {
                  /* Handle error */
            }
                  f = fopen(file, "w");
                  if (NULL = f) {
                        /* Handle error */
```

Time Of Check – Time Of Use (TOCTOU)[۱۲۵]

Race Window[۱۲۶]

```
                }

                /* Write to file */
                if (fclose(f) == EOF) {
                        /* Handle error */
                }
            }
        }
}
```

در قطعه کد بالا، پرونده با استفاده از نشانوند w برای نوشتن باز می‌شود. این نشانوند باعث می‌شود که اگر در پرونده بازشده محتویاتی وجود داشته باشد به‌طور کامل از بین برود. برای جلوگیری از بروز این مشکل ابتدا موجودیت پرونده با استفاده از باز کردن پرونده با نشانوند r بررسی می‌شود. چنانچه که پرونده موجود نبود عملیات باز کردن پرونده با نشانوند w انجام می‌شود.

در این حالت شخص حمله‌کننده در بین دو تابع fopen() می‌تواند محل فراخوانی پرونده را به پرونده‌ای دیگر اشاره داده و باعث پاک شدن آن شود.

قطعه کد زیر نمونه اصلاح‌شده قطعه کد بالا است.

```
void open_some_file(const char *file) {
        FILE *f = fopen(file, "wx");
        if (NULL != f) {
                /* FILE exists, handle error */
        }
        /* Write to file */
        if (fclose(f) == EOF) {
                /* Handle error */
        }
}
```

در این قطعه کد، از نشانوند wx که در C++11 اضافه‌شده است برای باز کردن پرونده استفاده می‌شود که باعث می‌شود در صورت موجودیت پرونده، از باز کردن آن خودداری کند.

در سیستم‌عامل لینوکس می‌توان از پرچم‌های O_CREAT و O_EXCL در تابع open() استفاده کرد که در صورت موجودیت پرونده، از باز کردن آن خودداری می‌کنند.

## ۹-۵ رمزنگاری داده‌های ورودی و خروجی

در سیر برنامه نیاز است که داده‌هایی به برنامه وارد و یا خارج شوند اما نیاز است که از صحت این داده‌ها اطمینان حاصل کرد. همچنین برخی از این داده‌ها همانند مشخصات اتصال به بانک اطلاعاتی نباید در دسترس اشخاص تأیید نشده قرار گرفتند.

یکی از روش‌های مناسب برای حافظت از داده‌های ورودی و خروجی، رمزنگاری آن‌هاست.

در این میان رمزنگاری با استفاده از کلید متقارن یکی از راهکارهای مناسب است چراکه این اطلاعات عمدتاً به‌صورت مداوم رمزنگاری و رمزگشایی می‌شوند و تمامی این فرایند توسط یک برنامه انجام می‌گیرد.

ازجمله مهم‌ترین اطلاعاتی که نیاز است در برنامه به‌صورت رمزنگاری‌شده ذخیره و فراخوانی شوند می‌توان به تنظیمات برنامه، اطلاعات اتصال به بانک اطلاعاتی، اطلاعات پروانه برنامه، مشخصات شخصی و بانکی کاربر است.

در این قسمت از رمزنگاری AES استفاده می‌کنیم. این رمزنگاری از ۳ نوع کلید ۱۲۸ بیتی، ۱۹۲ بیتی و ۲۵۶ بیتی پشتیبانی می‌کند که با افزایش سایز کلید امنیت نیز افزایش یافته اما رمزنگاری و رمزگشایی نیز سنگین‌تر می‌شود و نیازمند منابع و زمان بیشتری است. متد رمزنگاری CBC در AES به ما امکان رمزنگاری و رمزگشایی را با استفاده از key و iv[۱۲۷] می‌دهد.

AES بر مبنای بایت کارکرده و بایت‌ها را رمزنگاری می‌کند، حال این بایت‌ها می‌توانند بایت‌های یک پرونده یا یک‌رشته تبدیل‌شده به بایت یا یک شیء و یا نوع دیگری باشند.

قطعه کد زیر نمونه‌ای از رمزنگاری متقارن با استفاده از کتابخانه Crypto++ و با استفاده از مد AES CBC با کلید ۱۲۸ بیتی است:

```
byte key[CryptoPP::AES::DEFAULT_KEYLENGTH], iv[CryptoPP::AES::BLOCKSIZE];
menset(key, 0x00, CryptoPP::AES::DEFAULT_KEYLENGTH);
menset(iv, 0x00, CryptoPP::AES::BLOCKSIZE);
std::string plaintext = "Test";
std::string ciphertext;
std::string decryptedtext;
CryptoPP::AES::Encryption aesEncryption(key,
CryptoPP::AES::DEFAULT_KEYLENGTH);
CryptoPP::CBC_Mode_External1Cipher::Encryption
cbcEncryption(aesEncryption, iv);
CryptoPP::StreamTransformationFilter stfEncryptor(cbcEncryption, new
```

[۱۲۷] Initialization Vector

```
CryptoPP::StringSink(ciphertext));
sthncryptor.Put(reinterpret_cast<const unsigned
char*>(plaintext.c_str()), plaintext.length() + 1);
sthncryptor.MessageEnd();

CryptoPP::AES::Decryption aesDecryption(key,
CryptoPP::AES::DEFAULT_KEYLENGTH);
CryptoPP::CBC_Mode_External1Cipher::Decryption
cbcDecryption(aesDecryption, iv);
CryptoPP::StreamTransformationFilter stfDecryptor(cbcDecryption, new
CryptoPP::StringSink(decryptedtext));
sthecryptor.Put(reinterpret_cast<const unsigned
char*>(ciphertext.c_str()), ciphertext.size());
sthecryptor.MessageEnd();
```

در این قطعه کد ابتدا key و iv ساخته‌شده و سپس با استفاده از آن‌ها کلمه Test رمزنگاری می‌شود و سپس از با استفاده از همان key و iv عبارت رمزنگاری‌شده بازگردانی می‌شود.

باید توجه داشت که key و iv در هر بار اجرا ساخته‌شده و سپس از بین می‌روند لذا برای حفظ آن‌ها، باید آن‌ها را درون کد برنامه ذخیره کرد و یا آن‌ها را با استفاده از عبارت رشته‌ای، عددی و یا یک تابع بازسازی کرد.

## ۹-۶ ایمنی ارتباطات شبکه‌ای

ارتباطات شبکه‌ای بر روی هر بستری که باشند نیاز به ایمنی دارند چراکه این دسته از ارتباطات امکان شنود[۱۲۸] دارند و ممکن است از مبدأ به مقصد با انجام شنود اطلاعات ارسالی شنود شده و به نشر اطلاعات منجر شود. همچنین ممکن است اطلاعات در میانه راه دست‌کاری[۱۲۹] شده و تغییر یابد.

در این میان راهکارهای متفاوتی برای جلوگیری از این امر است.

اگر هدف صرفاً جلوگیری از دست‌کاری داده باشد، می‌توان با استفاده از ساخت هش از اطلاعات ارسالی صحت درستی آن‌ها را تأیید کرد.

یکی از رمزنگاری‌های مناسبت ارتباطات شبکه‌ای، رمزنگاری نامتقارن است.

---

[۱۲۸] Sniff

[۱۲۹] Tamper



مرکز مدیریت امداد و هماهنگی عملیات در حوادث رایانه ای

با توجه به کاربرد کلید عمومی در این نوع رمزنگاری، این امکان به برنامه‌نویس داده می‌شود تا کلید عمومی را اعلان عمومی کند و در اختیار عموم قرار دهد چراکه داشتن کلید عمومی ریسک امنیتی محسوب نمی‌شود.

همچنین می‌توان در هر اتصال یک کلید عمومی و خصوصی خاص برای همان ارتباط ساخته و بعد از اتمام ارتباط آن را از بین برد.

قطعه کد زیر، نمونه‌ای از رمزنگاری با متد کلید عمومی با استفاده از الگوریتم RSA و کتابخانه Crypto++ است:

```
CryptoPP::AutoSeededRandomPool rng;
CryptoPP::InvertibleRSAFunction privkey;
privkey.Initialize(rng, 1024);
CryptoPP::Base64Encoder privkeysink(new
CryptoPP::FileSink("c:\\privkey.txt"));
privkey.DEREncode(privkeysink);
privkeysink.MessageEnd();
CryptoPP::RSAFunction pubkey(privkey);
CryptoPP::Base64Encoder pubkeysink(new
CryptoPP::FileSink("c:\\pubkey.txt"));
pubkey.DEREncode(pubkeysink);
pubkeysink.MessageEnd();
```

در قطعه کد بالا یک کلید عمومی و یک کلید خصوصی 1024 بیتی ساخته‌شده و به‌صورت Base64 به‌صورت رشته‌ای درآمده و درون پرونده‌های privkey.txt و pubkey.txt ذخیره می‌کند.

```
CryptoPP::AutoSeededRandomPool rng;
CryptoPP::ByteQueue bytes;
CryptoPP::FileSource file("c:\\pubkey.txt", true, new
CryptoPP::Base64Decoder);
file.TransferTo(bytes);
bytes.MessageEnd();
CryptoPP::RSA::PublicKey pubKey;
pubKey.Load(bytes);
std::string plain = "Test", cipher;
CryptoPP::RSAES_OAEP_SHA_Encryptor e(pubKey);
CryptoPP::StringSource ss1(plain, true,
      new CryptoPP::PK_EncryptorFilter(rng, e,
      new CryptoPP::StringSink(cipher)));
```

قطعه کد بالا با استفاده از متد OAEP و SHA و کلید عمومی ساخته‌شده، عبارت Test را رمزنگاری کرده و درون cipher قرار می‌دهد.

```
CryptoPP::ByteQueue bytes;
CryptoPP::FileSource file("c:\\privkey.txt", true, new
CryptoPP::Base64Decoder);
file.TransferTo(bytes);
bytes.MessageEnd();
CryptoPP::RSA::PrivateKey privateKey;
privateKey.Load(bytes);
std::string recovered;
```

```
CryptoPP::RSAES_OAEP_SHA_Decryptor d(privateKey);
CryptoPP::stringSource ss2(cipher, true,
      new CryptoPP::PK_DecryptorFilter(rng, d,
      new CryptoPP::Stringsink(recovered)
      )
      );
```

قطعه کد بالا، با استفاده از متد OAEP و SHA و کلید خصوصی ساخته‌شده، عبارت رمزنگاری‌شده را بازگردانی می‌کند و درون recovered قرار می‌دهد.

## ۹-۷ انتقال اطلاعات بین سیستمی

معماری‌های مختلف سیستم‌ها از ترتیب بایت[۱۳۰]های متفاوتی استفاده می‌کنند که یا little endian و یا big endian است. به‌عنوان‌مثال IA-32 که از رایج‌ترین معماری‌های رایانه‌های امروزی است از little endian استفاده کرده اما PowerPC از big endian استفاده می‌کند. همچنین پروتکل‌هایی همانند TCP/IP نیز از big endian در برقراری ارتباطات خود استفاده می‌کنند.

این تفاوت‌ها حتی ممکن است در تعداد بیت‌ها در هر بایت، اعداد اعشاری، بیت‌های ساختاری و ویژگی‌های دیگر نیز باشد.

این تفاوت باعث می‌شود که نحوه پردازش اطلاعات باینری در ساختارهای مختلف، متفاوت باشد که درنتیجه اطلاعات در دو ساختار متفاوت یکسان نمی‌باشند.

در ارتباطات شبکه‌ای می‌توان از توابع ntohs(), htons(), htonl() برای تبدیل ترتیب بایت‌ها به big endian استفاده کرد. این توابع در سیستم‌هایی که معماری آن‌ها big endian باشد بی‌اثر است.

همچنین در زمان فراخوانی داده‌ها باید از توابع جامع استفاده کرد.

توابع fread() و fwrite() از رایج‌ترین توابعی هستند که باعث بروز مشکل در سیستم‌های مختلف می‌شوند و در صورتی نیاز به اجرای برنامه بر روی سیستم‌هایی با معماری متفاوت است باید از جایگزین این توابع استفاده کرد.

---

[۱۳۰] Byte Ordering

# ۱۰ نکات برنامه‌نویسی ایمن و تدافعی

## ۱۰-۱ نکات نگارشی و دستوری

۱- عدم استفاده از نام‌های رزرو شده برای نام‌گذاری

۲- عدم استفاده از namespace بدون نام در هدرها

۳- عدم تعریف دو یا چند ساختار ناسازگار[۱۳۱] از یک شیء یا تابع

۴- تابع‌های تعریف‌شده با [[noreturn]] باید از نوع void بوده و بدون بازگشت باشند.

۵- توابع تخریب‌کننده و آزاد ساز نباید باعث وجود خطا در برنامه شوند و باید خطاها کنترل‌شده و یا از noexcept در آن‌ها استفاده شود.

۶- لزوم بازگشت مقدار در هر شرایطی در توابعی که از نوع بازگشت‌پذیر می‌باشند.

۷- توابع رونویسی شده[۱۳۲] اپراتورهای پسوندی[۱۳۳] افزایش[۱۳۴] و کاهش[۱۳۵] باید یک شیء پایدار برگردانند.

۸- عدم استفاده از پرانتز باز و بسته در انتهای نام شیء ایجادشده از ساختارهایی جز تابع

۹- عدم استفاده const و violate به‌صورت اشاره‌ای

۱۰- عدم violate سازی ساختار ایستا

۱۱- توجه به مکان و محدوده تعریف اشیاء

---

[۱۳۱] Incompatible

[۱۳۲] Overloaded

[۱۳۳] Postfix

[۱۳۴] Increment

[۱۳۵] Decrement

مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

۱۲- عدم مقایسه داده‌های لایه‌ای[۱۳۶]

۱۳- عدم استفاده از UTF-16 به‌جز زمان فراخوانی توابع مربوط به سیستم‌عامل

۱۴- عدم مقداردهی در ساختار اصلی ساختارهای شرطی و انتخابی

۱۵- عدم دسترسی و اعمال تغییرات بر روی بیت‌های ساختاری شیء (همانند null char در عبارت‌های رشته‌ای)

۱۶- استفاده از offsetof() فقط بر روی گونه[۱۳۷] و اعضای معتبر

۱۷- عدم پاک‌سازی و آزادسازی آرایه با استفاده از اشاره‌گری که از جنس آرایه نیست و به آن اشاره می‌کند

۱۸- عدم ارسال ارجاع[۱۳۸] و یا گونه غیرقابل کپی به va_start()

۱۹- عدم فراخوانی va_arg() یا va_list() با مقدار نامشخص

۲۰- عدم ارسال مقدار خارج از محدوده به ساختار شمارشی[۱۳۹] (ارسال مقدار خارج از محدوده باعث برگرداندن مقدار نامعلوم و رفتار تعریف‌نشده می‌شود)

۲۱- عدم اضافه و کم کردن عدد صحیح به اشاره‌گری که به شیء ای جز آرایه اشاره می‌کند

۲۲- عدم قالب‌گیری[۱۴۰] اشاره‌گر به نوع محدودتر

۲۳- برای تبدیل اشاره‌گر به عدد صحیح و عدد صحیح به اشاره‌گر باید از uintptr_t استفاده کرد.

۲۴- عدم ارسال عبارت رشته‌ای بدون به پایان رساننده تهی[۱۴۱] به توابعی که ورودی عبارت رشته‌ای دارند

۲۵- بستن پرونده بلافاصله بعد از اتمام نیاز

۲۶- عدم دسترسی به پرونده بسته‌شده

۲۷- لزوم مدیریت تمامی خطاها

---

Padding Data[۱۳۶]

Type[۱۳۷]

Reference[۱۳۸]

Enumeration[۱۳۹]

Cast[۱۴۰]

Null Terminator[۱۴۱]

مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

۲۸- دقت به عدم نشر اطلاعات در زمان مدیریت خطاها

۲۹- عدم استفاده از std::terminate(),std::abort(),std::_Exit()

۳۰- عدم استفاده از setjmp() و longjmp()

۳۱- عدم کپی از شیء FILE

۳۲- عدم شیف یک عبارت[۱۴۲] با تعداد بیت مفنی و یا تعداد بیت بزرگ‌تر مساوی بیت‌های موجود در عملوند[۱۴۳]

۳۳- مقادیر ارسالی به fsetpos() فقط و فقط باید مقادیر بازگشتی از fgetpost() باشند.

۳۴- دستورات مربوط به مدیریت خطا خود نباید خطا ایجاد کنند.

۳۵- بازنشانی[۱۴۴] رشته‌ها در صورت عدم موفقیت fgets() و fgetws()

۳۶- توابع fgets() و fgetws() الزاماً در زمان موفقیت رشته غیر تهی بازنمی‌گردانند. این دو تابع در زمان موفقیت مقدار s را بازمی‌گردانند و اگر زمان پایان زندگی[۱۴۵] آن‌ها فرارسد و هیچ کاراکتری از درون آرایه خوانده‌نشده باشد، محتوای آرایه بدون تغییر مانده و اشاره‌گر تهی بازمی‌گردد

۳۷- اطمینان از وجود فضای کافی در حافظه برای ساختار و مقادیر شیء در زمان ساخت اشیاء و پردازش بر روی آن‌ها

۳۸- اطمینان از وجود ساختار مناسب بعد از انجام عملیات ریاضی

۳۹- استفاده از توابع تخلیه سازی بین توابع خواندن از پرونده و نوشتن بر روی پرونده در صورت استفاده پشت سر هم این دو تابع بر روی یک شیء

۴۰- عدم پاک‌سازی شیء چندریختی[۱۴۶] بدون تخریب‌کننده مجازی[۱۴۷]

۴۱- فراخوانی تخریب‌کننده تابع قبل از آزادسازی حافظه

---

Expression[۱۴۲]

Operand[۱۴۳]

Reset[۱۴۴]

End-OF-Life[۱۴۵]

Polymorphic[۱۴۶]

Virtual[۱۴۷]

مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

۴۲- رشته‌های لفظی[۱۴۸] و رشته‌های مقداردهی شده به اشاره‌گر کاراکتری باید پایدار مدنظر گرفته شوند و از اعمال تغییرات بر روی آن‌ها جلوگیری کرد.

۴۳- در صورت نیاز به مقداردهی و یا تبدیل داده کاراکتری با علامت[۱۴۹] به نوع با علامت بزرگ‌تر، نیاز است که ابتدا داده کاراکتری به نوع بدون علامت[۱۵۰] قالب‌گیری شود.

۴۴- عدم ویرایش و تغییر namespace های استاندارد

۴۵- عدم استفاده از اعداد اعشاری به‌عنوان شمارنده‌های حلقه

۴۶- عدم ذخیره، ارجاع و استفاده از اشاره‌گرهای بازگردانده شده از توابع ()asctime, ()getenv ()strerror, ()setlocale, ()localeconv

۴۷- عدم برش اشیاء مشتق شده

۴۸- عدم ارسال مقادیر نامعتبر به تابع ()asctime

۴۹- عدم تعریف چندباره یک شیء

۵۰- نشانوندهای ارسالی به توابع رسیدگی‌کننده به کاراکتر باید قابل‌ارائه به‌صورت کاراکتر بدون علامت باشند. این دسته از توابع به‌صورت زیر هستند:

| | | | |
|---|---|---|---|
| isalnum() | isalpha() | isascii()XSI | isblank() |
| iscntrl() | isdigit() | isgraph() | islower() |
| isprint() | ispunct() | isspace() | isupper() |
| isxdigit() | toascii()XSI | toupper() | tolower() |

۵۱- عدم استفاده از تابع ()memcmp برای مقایسه اعداد اعشاری

۵۲- عدم تعریف توابع variadic با ساختار C در C++، توابع variadic توابعی هستند که به تعداد متغیری نشانوند می‌گیرند.

---

Literal[۱۴۸]

Signed[۱۴۹]

Unsigned[۱۵۰]

۵۳- مقداردهی errno به صفر قبل از فراخوانی تابعی که errno را مقداردهی می‌کند و بازرسی errno فقط و فقط بعد از این‌که تابع مقداری مبنی بر شکست بازگرداند. این توابع در پیوست قسمت ۴ به نمایش درآمده‌اند.

۵۴- عدم استفاده از اپراتور new در انواعی که alingas در آن‌ها استفاده‌شده و استفاده از aligned_alloc به‌جای آن

۵۵- عدم استفاده از #ifdef, #define و #include در فراخوانی ماکروهای شبه تابع

۵۶- تمامی کنترل‌کننده‌های خروج باید بازگشت عادی داشته باشند.

۵۷- عدم دسترسی به اشیاء مشترک در کنترل‌کننده‌های سیگنال

۵۸- عدم فراخوانی سیگنال درون کنترل‌کننده قطع سیگنال

۵۹- کنترل‌کننده سیگنال باید یک تابع ساده باشد.

۶۰- عدم استفاده از سیگنال در برنامه‌های چند نخی

۶۱- عدم فراخوانی تابع‌های مجازی در سازنده[۱۵۱] و تخریب‌کننده[۱۵۲] توابع

۶۲- تمامی سازنده‌های اشیاء باصفت static نباید ایجاد خطا[۱۵۳] کنند.

۶۳- عدم ویرایش alignment اشیاء با فراخوانی realloc()

۶۴- اعمال کپی حافظه اصلی صرفاً باید باعث تغییر مقصد شوند.

۶۵- لزوم شناخت بازه و دامنه توابع ریاضی و خطاهای مربوط به‌آن‌ها، این موارد در پیوست قسمت ۵ به نمایش درآمده‌اند.

۶۶- عدم بازگشت از کنترل‌کننده سیگنال اعتراض محاسباتی

۶۷- عدم فراخوانی توابع putwc(), getwc(), putc(), getc() با استفاده از نشانوندهای جریانی[۱۵۴]ای که امکان بروز خطا و یا نتیجه غیرقابل‌اعتماد در آن‌هاست

۶۸- لزوم شناخت و مدیریت خطاهای توابع رایج زبان C/C++، این توابع به همراه خطاهایشان در پیوست قسمت ۶ به نمایش درآمده‌اند.

---

[۱۵۱] Constructor

[۱۵۲] Destructor

[۱۵۳] Exception

[۱۵۴] Stream

مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

۶۹- استفاده از براکت‌های باز و بسته [] بدون مقدار در زمان تعریف آرایه انعطاف‌پذیر[۱۵۵] در یک ساختار

۷۰- عدم تعریف شناسه با طبقه‌بندی‌های ارتباط متناقض، جدول زیر بیان‌گر این روابط است:

| | | Second | | |
|---|---|---|---|---|
| | | static | No linkage | extern |
| First | static | Internal | Undefined | Internal |
| | No linkage | Undefined | No linkage | External |
| | extern | Undefined | Undefined | External |

۷۱- فراخوانی توابع با تعداد و گونه صحیح نشانوندها

۷۲- عدم فراخوانی توابع زیر با مقادیر موجود در هدر complex.h:

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| atan2() | erf() | fdim() | fmin() | ilogb() | llround() | logb() | nextafter() | rint() | tgamma() |
| cbrt() | erfc() | floor() | fmod() | ldexp() | log10() | lrint() | nexttoward() | round() | trunc() |
| ceil() | exp2() | fma() | frexp() | lgamma() | log1p() | lround() | remainder() | scalbn() | |
| copysign() | expm1() | fmax() | hypot() | llrint() | log2() | nearbyint() | remquo() | scalbln() | |

۷۳- عدم دسترسی به متغییر با استفاده از اشاره‌گر از گونه ناسازگار

۷۴- عدم مقایسه داده‌های ساختاری

۷۵- عدم دسترسی به شیء با صرف volatile از طریق ارجاع بدون volatile

۷۶- عدم استفاده از اپراتورهای محاسباتی پیشوندی و پسوندی در نشانوند توابع

۷۷- عدم تفاضل و یا مقایسه دو اشاره‌گری که به یک آرایه یکسان اشاره نمی‌کنند

۷۸- عدم اضافه و یا کم کردن عدد صحیح مدرج[۱۵۶] از اشاره‌گر

---

[۱۵۵] Flexible

[۱۵۶] Scaled

۷۹- ساختارهایی که شامل آرایه انعطاف‌پذیر می‌باشند باید به‌صورت پویا[۱۵۷] کپی و تخصیص حافظه شوند.

۸۰- آزادسازی حافظه فقط باید بر روی تخصیص‌های پویا انجام گیرد.

۸۱- عدم انجام عملیات بر روی پرونده‌های سیستمی سیستم‌عامل

۸۲- عدم استفاده از توابعی که ممکن است باعث حالت پیشی گرفتن در برنامه شوند. این دسته از توابع به همراه راهکار جایگزین در جدول زیر به نمایش درآمده‌اند:

| Functions | Remediation |
|---|---|
| rand(), srand() | MSC30-C. Do not use the rand() function for generating pseudorandom numbers |
| getenv(), getenv_s() | ENV34-C. Do not store pointers returned by certain functions |
| strtok() | strtok_s() in C11 Annex K<br>strtok_r() in POSIX |
| strerror() | strerror_s() in C11 Annex K<br>strerror_r() in POSIX |
| asctime(), ctime(), localtime(), gmtime() | asctime_s(), ctime_s(), localtime_s(), gmtime_s() in C11 Annex K |
| setlocale() | Protect multithreaded access to locale-specific functions with a mutex |
| ATOMIC_VAR_INIT, atomic_init() | Do not attempt to initialize an atomic variable from multiple threads |
| tmpnam() | tmpnam_s() in C11 Annex K<br>tmpnam_r() in POSIX |
| mbrtoc16(), c16rtomb(), mbrtoc32(), c32rtomb() | Do not call with a null mbstate_t * argument |

۸۳- عدم الحاق نخی که قبلاً الحاق[۱۵۸] شده و یا جدا کردن[۱۵۹] نخی که قبلاً جداشده

۸۴- عدم ارجاع مضاعف به یک متغییر atomic در یک عبارت

۸۵- لزوم آزادسازی فضای مختص نخ[۱۶۰] ساخته‌شده با tss_create() با استفاده از tss_delete() و یا ترکیب توابع free(tss_get())

۸۶- لزوم توجه به زمان حیات اشیاء مشترک بین نخ‌ها

۸۷- عدم فراخوانی توابع ناهمگن درون کنترل‌کننده سیگنال

---

[۱۵۷] Dynamic

[۱۵۸] Join

[۱۵۹] Detach

[۱۶۰] Thread-Specific Storage

۸۸- اطمینان از این‌که توابع کتابخانه‌ای باعث بروز اشاره‌گر نامعتبر نمی‌شوند. توابع اشاره‌شده در زیر، اشاره‌گر و عدد صحیح را به‌عنوان ورودی دریافت می‌کنند که اشاره‌گر باید به مکان صحیحی در حافظه اشاره کند و عدد صحیح واردشده باید بازه‌ای از حافظه باشد که شامل المان‌های معتبری در حافظه هستند.

| | | | |
|---|---|---|---|
| fgets() | fgetws() | mbstowcs()[1] | wcstombs()[1] |
| mbrtoc16()[2] | mbrtoc32()[2] | mbsrtowcs()[1] | wcsrtombs()[1] |
| mbtowc()[2] | mbrtowc()[1] | mblen() | mbrlen() |
| memchr() | wmemchr() | memset() | wmemset() |
| strftime() | wcsftime() | strxfrm()[1] | wcsxfrm()[1] |
| strncat()[2] | wcsncat()[2] | snprintf() | vsnprintf() |
| swprintf() | vswprintf() | setvbuf() | tmpnam_s() |
| snprintf_s() | sprintf_s() | vsnprintf_s() | vsprintf_s() |
| gets_s() | getenv_s() | wctomb_s() | mbstowcs_s()[3] |
| wcstombs_s()[3] | memcpy_s()[3] | memmove_s()[3] | strncpy_s()[3] |
| strncat_s()[3] | strtok_s()[2] | strerror_s() | strnlen_s() |
| asctime_s() | ctime_s() | snwprintf_s() | swprintf_s() |
| vsnwprintf_s() | vswprintf_s() | wcsncpy_s()[3] | wmemcpy_s()[3] |
| wmemmove_s()[3] | wcsncat_s()[3] | wcstok_s()[2] | wcsnlen_s() |
| wcrtomb_s() | mbsrtowcs_s()[3] | wcsrtombs_s()[3] | memset_s()[4] |

۱- دو اشاره‌گر و یک عدد صحیح دریافت می‌کند اما عدد صحیح بیانگر شمار المان‌های بافر خروجی است.

۲- دو اشاره‌گر و یک عدد صحیح دریافت می‌کند اما عدد صحیح بیانگر شمار المان‌های بافر ورودی است.

۳- دو اشاره‌گر و دو عدد صحیح دریافت می‌کند که هرکدام بیانگر شمار المان‌های یک اشاره‌گر می‌باشند.

۴- یک اشاره‌گر و دو عدد صحیح دریافت می‌کند که عدد صحیح اول بیانگر تعداد بایت‌های موجود در بافر و عدد صحیح دوم تعداد بایت‌ها برای نوشته شدن درون بافر است.

۸۹- عدم مقداردهی اشاره‌گر restrict با اشاره‌گر restrict دیگر

۹۰- لزوم تقدم و استفاده از معادل C++ به‌جای توابع کتابخانه‌ای زیر موجود در زبان C:

| C Standard Library Function | C++ Equivalent Functionality |
|---|---|
| std::memset() | Class constructor |
| std::memcpy()<br>std::memmove()<br>std::strcpy() | Class copy constructor or operator=() |
| std::memcmp()<br>std::strcmp() | operator<(), operator>(), operator==(), or operator!=() |

۹۱- عدم ارسال اشیاء با ترتیب غیراستاندارد بین مرزهای اجرایی[۱۶۱]

ویژگی‌های کلاس استاندارد عبارت‌اند از:

۱- توابع مجازی ندارند.

۲- برای تمامی اعضای غیر ایستا سطح دسترسی یکسانی دارند.

۳- کلاس‌های پایه‌ای از یک نوع، عضو غیرایستا ندارند.

۴- اعضای داده‌ای غیر ایستا فقط در یک کلاس از سلسله‌مراتب[۱۶۲] حضور دارند.

۵- به‌صورت بازگشتی، فاقد عضو داده‌ای غیر ایستایی با ترتیب غیراستاندارد می‌باشند.

۹۲- اطمینان از عدم از دست رفتن و یا تغییر ناخواسته در زمان تبدیل و یا فراخوانی توابع بر روی عدد صحیح

## ۱۰-۲ نکات مختص سیستم‌عامل خانواده Unix

۱- عدم استفاده از تابع vfork() و استفاده از تابع fork() به‌جای آن

---

[۱۶۱] Execution Boundaries

[۱۶۲] Hierarchy

۲- تابع readlink() در نشانوند دوم خود یک بافر را دریافت می‌کند که به پایان رساننده تهی توجه نداشته و فقط شماری از کاراکترهای نوشته‌شده را بازمی‌گرداند.

۳- عدم استفاده از سیگنال‌ها برای به پایان رساندن نخ‌ها

۴- عدم استفاده از نخ‌هایی که امکان لغو شدن به‌صورت ناهمگام دارند

۵- عدم از قفل درآوردن و یا تخریب mutex نخ‌های دیگر

۶- لزوم شناخت و مدیریت خطای توابع کتابخانه‌ای سیستم‌عامل که ۳ تابع اصلی این دسته به‌صورت زیر هستند:

| Function | Successful Return | Error Return | errno |
|---|---|---|---|
| fmemopen() | Pointer to a FILE object | NULL | ENOMEM |
| open_memstream() | Pointer to a FILE object | NULL | ENOMEM |
| posix_memalign() | 0 | Nonzero | Unchanged |

۷- عدم استفاده بیش از یک mutex بر روی تغییر وضعیت در زمان استفاده از توابع pthread_cond_wait() و pthread_cond_timewait()

۸- اطمینان از بازگردانی دسترسی‌ها به سطح اولیه در زمان فراخوانی تابع setuid()

۹- عدم به وجود آوردن حالت پیشی گرفتن زمان بازرسی زمان استفاده در زمان بازرسی موجودیت Symbol Link، راهکار مناسب برای بازرسی موجودیت Symbol Link به‌صورت زیر است:

۱- فراخوانی lstat() بر روی نام پرونده

۲- فراخوانی open() برای باز کردن پرونده

۳- فراخوانی fstat() بر روی واصف[۱۶۳] پرونده بازگردانی شده توسط open()

۴- مقایسه اطلاعات بازگردانی شده توسط lstat() و fstat() برای اطمینان از این‌که پرونده‌ها یکسان‌اند.

۱۰- فراخوانی putenv() با نشانوند تغییر و یا آرایه دارای صفت static و یا فضای پویا تخصیص‌یافته از Heap یا استفاده از setenv() به‌جای آن

[۱۶۳] Descriptor

# ۱۱ پیوست

## ۱۱-۱ خانواده printf

```
int printf( const char *format, ... );                                  (until C99)
int printf( const char *restrict format, ... );                         (since C99)
int fprintf( FILE *stream, const char *format, ... );                   (until C99)
int fprintf( FILE *restrict stream, const char *restrict format, ... );  (since C99)
int sprintf( char *buffer, const char *format, ... );                   (until C99)
int sprintf( char *restrict buffer, const char *restrict format, ... );  (since C99)
int snprintf( char *restrict buffer, int bufsz,
              const char *restrict format, ... );                       (since C99)
int printf_s(const char *restrict format, ...);                         (since C11)
int fprintf_s(FILE *restrict stream, const char *restrict format, ...);  (since C11)
int sprintf_s(char *restrict buffer, rsize_t bufsz,
              const char *restrict format, ...);                        (since C11)
int snprintf_s(char *restrict buffer, rsize_t bufsz,
               const char *restrict format, ...);                       (since C11)
```

۱۶۴

```
int wprintf( const wchar_t* format, ... );                              (since C95)
int fwprintf( FILE *stream, const wchar_t* format, ... );               (since C95)
int swprintf( wchar_t *buffer, size_t bufsz,
              const wchar_t* format, ... );                             (since C95)
int wprintf_s( const wchar_t *restrict format, ...);                    (since C11)
int fwprintf_s( FILE *restrict stream,
                const wchar_t *restrict format, ...);                   (since C11)
int swprintf_s( wchar_t *restrict buffer, rsize_t bufsz,
                const wchar_t* restrict format, ...);                   (since C11)
int snwprintf_s( wchar_t * restrict s, rsize_t n,
                 const wchar_t * restrict format, ...);                 (since C11)
```

۱۶۵

http://en.cppreference.com/w/c/io/fprintf[۱۶۴]

http://en.cppreference.com/w/c/io/fwprintf[۱۶۵]

```
int vprintf( const char *format, va_list vlist );                          (until C99)
int vprintf( const char *restrict format, va_list vlist );                 (since C99)
int vfprintf( FILE *stream, const char *format, va_list vlist );           (until C99)
int vfprintf( FILE *restrict stream, const char *restrict format,
              va_list vlist );                                             (since C99)
int vsprintf( char *buffer, const char *format, va_list vlist );           (until C99)
int vsprintf( char *restrict buffer, const char *restrict format,
              va_list vlist );                                             (since C99)
int vsnprintf( char *restrict buffer, int bufsz,
               const char *restrict format, va_list vlist );               (since C99)
int vprintf_s( const char *restrict format, va_list arg);                  (since C11)
int vfprintf_s( FILE *restrict stream, const char *restrict format,
                va_list arg);                                              (since C11)
int vsprintf_s( char *restrict buffer, rsize_t bufsz,
                const char *restrict format, va_list arg);                 (since C11)
int vsnprintf_s(char *restrict buffer, rsize_t bufsz,
                const char * restrict format, va_list arg);                (since C11)
```

۱۶۶

```
int scanf( const char         *format, ... );                              (until C99)
int scanf( const char *restrict format, ... );                             (since C99)
int fscanf( FILE         *stream, const char         *format, ... );       (until C99)
int fscanf( FILE *restrict stream, const char *restrict format, ... );     (since C99)
int sscanf( const char         *buffer, const char         *format, ... );  (until C99)
int sscanf( const char *restrict buffer, const char *restrict format, ... ); (since C99)
int scanf_s(const char *restrict format, ...);                             (since C11)
int fscanf_s(FILE *restrict stream, const char *restrict format, ...);     (since C11)
int sscanf_s(const char *restrict buffer, const char *restrict format, ...); (since C11)
```

۱۶۷

http://en.cppreference.com/w/c/io/vfprintf[۱۶۶]

http://en.cppreference.com/w/c/io/fscanf[۱۶۷]

## ۲-۱۱ atomic

| Typedef name | Full specialization |
|---|---|
| std::atomic_bool | std::atomic<bool> |

| Typedef name | Full specialization |
|---|---|
| std::atomic_char | std::atomic<char> |
| std::atomic_schar | std::atomic<signed char> |
| std::atomic_uchar | std::atomic<unsigned char> |
| std::atomic_short | std::atomic<short> |
| std::atomic_ushort | std::atomic<unsigned short> |
| std::atomic_int | std::atomic<int> |
| std::atomic_uint | std::atomic<unsigned int> |
| std::atomic_long | std::atomic<long> |
| std::atomic_ulong | std::atomic<unsigned long> |
| std::atomic_llong | std::atomic<long long> |
| std::atomic_ullong | std::atomic<unsigned long long> |
| std::atomic_char16_t | std::atomic<char16_t> |
| std::atomic_char32_t | std::atomic<char32_t> |
| std::atomic_wchar_t | std::atomic<wchar_t> |
| std::atomic_int_least8_t | std::atomic<std::int_least8_t> |
| std::atomic_uint_least8_t | std::atomic<std::uint_least8_t> |
| std::atomic_int_least16_t | std::atomic<std::int_least16_t> |
| std::atomic_uint_least16_t | std::atomic<std::uint_least16_t> |
| std::atomic_int_least32_t | std::atomic<std::int_least32_t> |
| std::atomic_uint_least32_t | std::atomic<std::uint_least32_t> |
| std::atomic_int_least64_t | std::atomic<std::int_least64_t> |
| std::atomic_uint_least64_t | std::atomic<std::uint_least64_t> |
| std::atomic_int_fast8_t | std::atomic<std::int_fast8_t> |
| std::atomic_uint_fast8_t | std::atomic<std::uint_fast8_t> |
| std::atomic_int_fast16_t | std::atomic<std::int_fast16_t> |
| std::atomic_uint_fast16_t | std::atomic<std::uint_fast16_t> |
| std::atomic_int_fast32_t | std::atomic<std::int_fast32_t> |
| std::atomic_uint_fast32_t | std::atomic<std::uint_fast32_t> |
| std::atomic_int_fast64_t | std::atomic<std::int_fast64_t> |
| std::atomic_uint_fast64_t | std::atomic<std::uint_fast64_t> |
| std::atomic_intptr_t | std::atomic<std::intptr_t> |
| std::atomic_uintptr_t | std::atomic<std::uintptr_t> |
| std::atomic_size_t | std::atomic<std::size_t> |
| std::atomic_ptrdiff_t | std::atomic<std::ptrdiff_t> |
| std::atomic_intmax_t | std::atomic<std::intmax_t> |
| std::atomic_uintmax_t | std::atomic<std::uintmax_t> |

مرکز مدیریت امداد و

۱۶۸

http://en.cppreference.com/w/cpp/atomic/atomic[۱۶۸]

## ۳-۱۱ ثابت‌های محافظ حافظه

| Constant/value | Description |
|---|---|
| **PAGE_EXECUTE**<br>0x10 | Enables execute access to the committed region of pages. An attempt to write to the committed region results in an access violation.<br><br>This flag is not supported by the **CreateFileMapping** function. |
| **PAGE_EXECUTE_READ**<br>0x20 | Enables execute or read-only access to the committed region of pages. An attempt to write to the committed region results in an access violation.<br><br>**Windows Server 2003 and Windows XP:** This attribute is not supported by the **CreateFileMapping** function until Windows XP with SP2 and Windows Server 2003 with SP1. |
| **PAGE_EXECUTE_READWRITE**<br>0x40 | Enables execute, read-only, or read/write access to the committed region of pages.<br><br>**Windows Server 2003 and Windows XP:** This attribute is not supported by the **CreateFileMapping** function until Windows XP with SP2 and Windows Server 2003 with SP1. |
| **PAGE_EXECUTE_WRITECOPY**<br>0x80 | Enables execute, read-only, or copy-on-write access to a mapped view of a file mapping object. An attempt to write to a committed copy-on-write page results in a private copy of the page being made for the process. The private page is marked as **PAGE_EXECUTE_READWRITE**, and the change is written to the new page.<br><br>This flag is not supported by the **VirtualAlloc** or **VirtualAllocEx** functions.<br><br>**Windows Vista, Windows Server 2003, and Windows XP:** This attribute is not supported by the **CreateFileMapping** function until Windows Vista with SP1 and Windows Server 2008. |
| **PAGE_NOACCESS**<br>0x01 | Disables all access to the committed region of pages. An attempt to read from, write to, or execute the committed region results in an access violation.<br><br>This flag is not supported by the **CreateFileMapping** function. |
| **PAGE_READONLY**<br>0x02 | Enables read-only access to the committed region of pages. An attempt to write to the committed region results in an access violation. If Data Execution Prevention is enabled, an attempt to execute code in the committed region results in an access violation. |
| **PAGE_READWRITE**<br>0x04 | Enables read-only or read/write access to the committed region of pages. If Data Execution Prevention is enabled, attempting to execute code in the committed region results in an access violation. |
| **PAGE_WRITECOPY**<br>0x08 | Enables read-only or copy-on-write access to a mapped view of a file mapping object. An attempt to write to a committed copy-on-write page results in a private copy of the page being made for the process. The private page is marked as **PAGE_READWRITE**, and the change is written to the new page. If Data Execution Prevention is enabled, attempting to execute code in the committed region results in an access violation.<br><br>This flag is not supported by the **VirtualAlloc** or **VirtualAllocEx** functions. |
| **PAGE_TARGETS_INVALID**<br>0x40000000 | Sets all locations in the pages as invalid targets for CFG. Used along with any execute page protection like **PAGE_EXECUTE**, **PAGE_EXECUTE_READ**, **PAGE_EXECUTE_READWRITE** and **PAGE_EXECUTE_WRITECOPY**. Any indirect call to locations in those pages will fail CFG checks and the process will be terminated. The default behavior for executable pages allocated is to be marked valid call targets for CFG.<br><br>This flag is not supported by the **VirtualProtect** or **CreateFileMapping** functions. |
| **PAGE_TARGETS_NO_UPDATE**<br>0x40000000 | Pages in the region will not have their CFG information updated while the protection changes for **VirtualProtect**. For example, if the pages in the region was allocated using **PAGE_TARGETS_INVALID**, then the invalid information will be maintained while the page protection changes. This flag is only valid when the protection changes to an executable type like **PAGE_EXECUTE**, **PAGE_EXECUTE_READ**, **PAGE_EXECUTE_READWRITE** and **PAGE_EXECUTE_WRITECOPY**. The default behavior for **VirtualProtect** protection change to executable is to mark all locations as valid call targets for CFG. |

| Constant/value | Description |
|---|---|
| **PAGE_GUARD**<br>0x100 | Pages in the region become guard pages. Any attempt to access a guard page causes the system to raise a **STATUS_GUARD_PAGE_VIOLATION** exception and turn off the guard page status. Guard pages thus act as a one-time access alarm. For more information, see Creating Guard Pages.<br><br>When an access attempt leads the system to turn off guard page status, the underlying page protection takes over.<br><br>If a guard page exception occurs during a system service, the service typically returns a failure status indicator.<br><br>This value cannot be used with **PAGE_NOACCESS**.<br><br>This flag is not supported by the **CreateFileMapping** function. |
| **PAGE_NOCACHE**<br>0x200 | Sets all pages to be non-cachable. Applications should not use this attribute except when explicitly required for a device. Using the interlocked functions with memory that is mapped with **SEC_NOCACHE** can result in an **EXCEPTION_ILLEGAL_INSTRUCTION** exception.<br><br>The **PAGE_NOCACHE** flag cannot be used with the **PAGE_GUARD**, **PAGE_NOACCESS**, or **PAGE_WRITECOMBINE** flags.<br><br>The **PAGE_NOCACHE** flag can be used only when allocating private memory with the **VirtualAlloc**, **VirtualAllocEx**, or **VirtualAllocExNuma** functions. To enable non-cached memory access for shared memory, specify the **SEC_NOCACHE** flag when calling the **CreateFileMapping** function. |
| **PAGE_WRITECOMBINE**<br>0x400 | Sets all pages to be write-combined.<br><br>Applications should not use this attribute except when explicitly required for a device. Using the interlocked functions with memory that is mapped as write-combined can result in an **EXCEPTION_ILLEGAL_INSTRUCTION** exception.<br><br>The **PAGE_WRITECOMBINE** flag cannot be specified with the **PAGE_NOACCESS**, **PAGE_GUARD**, and **PAGE_NOCACHE** flags.<br><br>The **PAGE_WRITECOMBINE** flag can be used only when allocating private memory with the **VirtualAlloc**, **VirtualAllocEx**, or **VirtualAllocExNuma** functions. To enable write-combined memory access for shared memory, specify the **SEC_WRITECOMBINE** flag when calling the **CreateFileMapping** function.<br><br>**Windows Server 2003 and Windows XP:** This flag is not supported until Windows Server 2003 with SP1. |

| Constant | Description |
|---|---|
| **PAGE_ENCLAVE_THREAD_CONTROL** | The page contains a thread control structure (TCS). |
| **PAGE_ENCLAVE_UNVALIDATED** | The page contents that you supply are excluded from measurement with the EEXTEND instruction of the Intel SGX programming model. |

۱۶۹

https://msdn.microsoft.com/en-us/library/windows/desktop/aa366786%28v=vs.85%29.aspx[۱۶۹]

## ۱۱-٤ توابع تأثیرگذار بر errno

توابعی که مقادیر بازگشتی از آن‌ها در زمان شکست به‌هیچ‌وجه نمی‌تواند در زمان موفقیت بازگردد:

| Function Name | Return Value | errno Value |
|---|---|---|
| ftell() | -1L | Positive |
| fgetpos(), fsetpos() | Nonzero | Positive |
| mbrtowc(), mbsrtowcs() | (size_t)(-1) | EILSEQ |
| signal() | SIG_ERR | Positive |
| wcrtomb(), wcsrtombs() | (size_t)(-1) | EILSEQ |
| mbrtoc16(), mbrtoc32() | (size_t)(-1) | EILSEQ |
| c16rtomb(), cr32rtomb() | (size_t)(-1) | EILSEQ |

توابعی که مقادیر بازگشتی از آن‌ها در زمان شکست نمی‌تواند مقداری در زمان موفقیت نیز باشد:

| Function Name | Return Value | errno Value |
|---|---|---|
| fgetwc(), fputwc() | WEOF | EILSEQ |
| strtol(), wcstol() | LONG_MIN or LONG_MAX | ERANGE |
| strtoll(), wcstoll() | LLONG_MIN or LLONG_MAX | ERANGE |
| strtoul(), wcstoul() | ULONG_MAX | ERANGE |
| strtoull(), wcstoull() | ULLONG_MAX | ERANGE |
| strtoumax(), wcstoumax() | UINTMAX_MAX | ERANGE |
| strtod(), wcstod() | 0 or ±HUGE_VAL | ERANGE |
| strtof(), wcstof() | 0 or ±HUGE_VALF | ERANGE |
| strtold(), wcstold() | 0 or ±HUGE_VALL | ERANGE |
| strtoimax(), wcstoimax() | INTMAX_MIN, INTMAX_MAX | ERANGE |

## ۵-۱۱ بازه و دامنه توابع ریاضی

| Function | Domain | Range | Pole |
|---|---|---|---|
| acos(x) | -1 <= x && x <= 1 | No | No |
| asin(x) | -1 <= x && x <= 1 | Yes | No |
| atan(x) | None | Yes | No |
| atan2(y, x) | x != 0 && y != 0 | No | No |
| acosh(x) | x >= 1 | Yes | No |
| asinh(x) | None | Yes | No |
| atanh(x) | -1 < x && x < 1 | Yes | Yes |
| cosh(x), sinh(x) | None | Yes | No |
| exp(x), exp2(x), expm1(x) | None | Yes | No |
| ldexp(x, exp) | None | Yes | No |
| log(x), log10(x), log2(x) | x >= 0 | No | Yes |
| log1p(x) | x >= -1 | No | Yes |
| ilogb(x) | x != 0 && !isinf(x) && !isnan(x) | Yes | No |
| logb(x) | x != 0 | Yes | Yes |
| scalbn(x, n), scalbln(x, n) | None | Yes | No |
| hypot(x, y) | None | Yes | No |
| pow(x,y) | x > 0 \|\| (x == 0 && y > 0) \|\| (x < 0 && y is an integer) | Yes | Yes |
| sqrt(x) | x >= 0 | No | No |
| erf(x) | None | Yes | No |
| erfc(x) | None | Yes | No |
| lgamma(x), tgamma(x) | x != 0 && ! (x < 0 && x is an integer) | Yes | Yes |
| lrint(x), lround(x) | None | Yes | No |
| fmod(x, y), remainder(x, y), remquo(x, y, quo) | y != 0 | Yes | No |

| Function | Domain | Range | Pole |
|---|---|---|---|
| nextafter(x, y), nexttoward(x, y) | None | Yes | No |
| fdim(x,y) | None | Yes | No |
| fma(x,y,z) | None | Yes | No |

## ۶-۱۱ خطاهای رایج توابع موجود در زبان C/C++

مرکز فرمند

| Function | Successful Return | Error Return |
|---|---|---|
| aligned_alloc() | Pointer to space | NULL |
| asctime_s() | 0 | Nonzero |
| at_quick_exit() | 0 | Nonzero |
| atexit() | 0 | Nonzero |
| bsearch() | Pointer to matching element | NULL |
| bsearch_s() | Pointer to matching element | NULL |
| btowc() | Converted wide character | WEOF |
| c16rtomb() | Number of bytes | (size_t)(-1) |
| c32rtomb() | Number of bytes | (size_t)(-1) |
| calloc() | Pointer to space | NULL |
| clock() | Processor time | (clock_t)(-1) |
| cnd_broadcast() | thrd_success | thrd_error |
| cnd_init() | thrd_success | thrd_nomem or thrd_error |
| cnd_signal() | thrd_success | thrd_error |
| cnd_timedwait() | thrd_success | thrd_timedout or thrd_error |
| cnd_wait() | thrd_success | thrd_error |
| ctime_s() | 0 | Nonzero |
| fclose() | 0 | EOF (negative) |
| fflush() | 0 | EOF (negative) |
| fgetc() | Character read | EOF[1] |
| fgetpos() | 0 | Nonzero, errno > 0 |
| fgets() | Pointer to string | NULL |
| fgetwc() | Wide character read | WEOF[1] |
| fopen() | Pointer to stream | NULL |
| fopen_s() | 0 | Nonzero |

| Function | Successful Return | Error Return |
|---|---|---|
| fprintf() | Number of characters (nonnegative) | Negative |
| fprintf_s() | Number of characters (nonnegative) | Negative |
| fputc() | Character written | EOF[2] |
| fputs() | Nonnegative | EOF (negative) |
| fputwc() | Wide character written | WEOF |
| fputws() | Nonnegative | EOF (negative) |
| fread() | Elements read | Elements read |
| freopen() | Pointer to stream | NULL |
| freopen_s() | 0 | Nonzero |
| fscanf() | Number of conversions (nonnegative) | EOF (negative) |
| fscanf_s() | Number of conversions (nonnegative) | EOF (negative) |
| fseek() | 0 | Nonzero |
| fsetpos() | 0 | Nonzero, errno > 0 |
| ftell() | File position | -1L, errno > 0 |
| fwprintf() | Number of wide characters (nonnegative) | Negative |
| fwprintf_s() | Number of wide characters (nonnegative) | Negative |
| fwrite() | Elements written | Elements written |
| fwscanf() | Number of conversions (nonnegative) | EOF (negative) |
| fwscanf_s() | Number of conversions (nonnegative) | EOF (negative) |
| getc() | Character read | EOF[1] |
| getchar() | Character read | EOF[1] |
| getenv() | Pointer to string | NULL |
| getenv_s() | Pointer to string | NULL |
| gets_s() | Pointer to string | NULL |
| getwc() | Wide character read | WEOF |

| Function | Successful Return | Error Return |
|---|---|---|
| getwc() | Wide character read | WEOF |
| getwchar() | Wide character read | WEOF |
| gmtime() | Pointer to broken-down time | NULL |
| gmtime_s() | Pointer to broken-down time | NULL |
| localtime() | Pointer to broken-down time | NULL |
| localtime_s() | Pointer to broken-down time | NULL |
| malloc() | Pointer to space | NULL |
| mblen(), s != NULL | Number of bytes | -1 |
| mbrlen(), s != NULL | Number of bytes or status | (size_t)(-1) |
| mbrtoc16() | Number of bytes or status | (size_t)(-1), errno == EILSEQ |
| mbrtoc32() | Number of bytes or status | (size_t)(-1), errno == EILSEQ |
| mbrtowc(), s != NULL | Number of bytes or status | (size_t)(-1), errno == EILSEQ |
| mbsrtowcs() | Number of non-null elements | (size_t)(-1), errno == EILSEQ |
| mbsrtowcs_s() | 0 | Nonzero |
| mbstowcs() | Number of non-null elements | (size_t)(-1) |
| mbstowcs_s() | 0 | Nonzero |
| mbtowc(), s != NULL | Number of bytes | -1 |
| memchr() | Pointer to located character | NULL |
| mktime() | Calendar time | (time_t)(-1) |
| mtx_init() | thrd_success | thrd_error |
| mtx_lock() | thrd_success | thrd_error |
| mtx_timedlock() | thrd_success | thrd_timedout or thrd_error |
| mtx_trylock() | thrd_success | thrd_busy or thrd_error |
| mtx_unlock() | thrd_success | thrd_error |
| printf_s() | Number of characters (nonnegative) | Negative |

| Function | Successful Return | Error Return |
|---|---|---|
| putc() | Character written | EOF[2] |
| putwc() | Wide character written | WEOF |
| raise() | 0 | Nonzero |
| realloc() | Pointer to space | NULL |
| remove() | 0 | Nonzero |
| rename() | 0 | Nonzero |
| setlocale() | Pointer to string | NULL |
| setvbuf() | 0 | Nonzero |
| scanf() | Number of conversions (nonnegative) | EOF (negative) |
| scanf_s() | Number of conversions (nonnegative) | EOF (negative) |
| signal() | Pointer to previous function | SIG_ERR, errno > 0 |
| snprintf() | Number of characters that would be written (nonnegative) | Negative |
| snprintf_s() | Number of characters that would be written (nonnegative) | Negative |
| sprintf() | Number of non-null characters written | Negative |
| sprintf_s() | Number of non-null characters written | Negative |
| sscanf() | Number of conversions (nonnegative) | EOF (negative) |
| sscanf_s() | Number of conversions (nonnegative) | EOF (negative) |
| strchr() | Pointer to located character | NULL |
| strerror_s() | 0 | Nonzero |
| strftime() | Number of non-null characters | 0 |
| strpbrk() | Pointer to located character | NULL |
| strrchr() | Pointer to located character | NULL |
| strstr() | Pointer to located string | NULL |
| strtod() | Converted value | 0, errno == ERANGE |
| strtof() | Converted value | 0, errno == ERANGE |

| Function | Successful Return | Error Return |
|---|---|---|
| strtoimax() | Converted value | INTMAX_MAX or INTMAX_MIN, errno == ERANGE |
| strtok() | Pointer to first character of a token | NULL |
| strtok_s() | Pointer to first character of a token | NULL |
| strtol() | Converted value | LONG_MAX or LONG_MIN, errno == ERANGE |
| strtold() | Converted value | 0, errno == ERANGE |
| strtoll() | Converted value | LLONG_MAX or LLONG_MIN, errno == ERANGE |
| strtoumax() | Converted value | UINTMAX_MAX, errno == ERANGE |
| strtoul() | Converted value | ULONG_MAX, errno == ERANGE |
| strtoull() | Converted value | ULLONG_MAX, errno == ERANGE |
| strxfrm() | Length of transformed string | >= n |
| swprintf() | Number of non-null wide characters | Negative |
| swprintf_s() | Number of non-null wide characters | Negative |
| swscanf() | Number of conversions (nonnegative) | EOF (negative) |
| swscanf_s() | Number of conversions (nonnegative) | EOF (negative) |
| thrd_create() | thrd_success | thrd_nomem or thrd_error |
| thrd_detach() | thrd_success | thrd_error |
| thrd_join() | thrd_success | thrd_error |
| thrd_sleep() | 0 | Negative |
| time() | Calendar time | (time_t)(-1) |
| timespec_get() | Base | 0 |
| tmpfile() | Pointer to stream | NULL |
| tmpfile_s() | 0 | Nonzero |
| tmpnam() | Non-null pointer | NULL |
| tmpnam_s() | 0 | Nonzero |
| tss_create() | thrd_success | thrd_error |

| Function | Successful Return | Error Return |
|---|---|---|
| tss_get() | Value of thread-specific storage | 0 |
| tss_set() | thrd_success | thrd_error |
| ungetc() | Character pushed back | EOF (see below) |
| ungetwc() | Character pushed back | WEOF |
| vfprintf() | Number of characters (nonnegative) | Negative |
| vfprintf_s() | Number of characters (nonnegative) | Negative |
| vfscanf() | Number of conversions (nonnegative) | EOF (negative) |
| vfscanf_s() | Number of conversions (nonnegative) | EOF (negative) |
| vfwprintf() | Number of wide characters (nonnegative) | Negative |
| vfwprintf_s() | Number of wide characters (nonnegative) | Negative |
| vfwscanf() | Number of conversions (nonnegative) | EOF (negative) |
| vfwscanf_s() | Number of conversions (nonnegative) | EOF (negative) |
| vprintf_s() | Number of characters (nonnegative) | Negative |
| vscanf() | Number of conversions (nonnegative) | EOF (negative) |
| vscanf_s() | Number of conversions (nonnegative) | EOF (negative) |
| vsnprintf() | Number of characters that would be written (nonnegative) | Negative |
| vsnprintf_s() | Number of characters that would be written (nonnegative) | Negative |
| vsprintf() | Number of non-null characters (nonnegative) | Negative |
| vsprintf_s() | Number of non-null characters (nonnegative) | Negative |
| vsscanf() | Number of conversions (nonnegative) | EOF (negative) |
| vsscanf_s() | Number of conversions (nonnegative) | EOF (negative) |
| vswprintf() | Number of non-null wide characters | Negative |
| vswprintf_s() | Number of non-null wide characters | Negative |
| vswscanf() | Number of conversions (nonnegative) | EOF (negative) |
| vswscanf_s() | Number of conversions (nonnegative) | EOF (negative) |

ی رایانه ای

| Function | Successful Return | Error Return |
|---|---|---|
| wcrtomb() | Number of bytes stored | (size_t)(-1) |
| wcschr() | Pointer to located wide character | NULL |
| wcsftime() | Number of non-null wide characters | 0 |
| wcspbrk() | Pointer to located wide character | NULL |
| wcsrchr() | Pointer to located wide character | NULL |
| wcsrtombs() | Number of non-null bytes | (size_t)(-1), errno == EILSEQ |
| wcsrtombs_s() | 0 | Nonzero |
| wcsstr() | Pointer to located wide string | NULL |
| wcstod() | Converted value | 0, errno == ERANGE |
| wcstof() | Converted value | 0, errno == ERANGE |
| wcstoimax() | Converted value | INTMAX_MAX or INTMAX_MIN, errno == ERANGE |
| wcstok() | Pointer to first wide character of a token | NULL |
| wcstok_s() | Pointer to first wide character of a token | NULL |
| wcstol() | Converted value | LONG_MAX or LONG_MIN, errno == ERANGE |
| wcstold() | Converted value | 0, errno == ERANGE |
| wcstoll() | Converted value | LLONG_MAX or LLONG_MIN, errno == ERANGE |
| wcstombs() | Number of non-null bytes | (size_t)(-1) |
| wcstombs_s() | 0 | Nonzero |
| wcstoumax() | Converted value | UINTMAX_MAX, errno == ERANGE |
| wcstoul() | Converted value | ULONG_MAX, errno == ERANGE |
| wcstoull() | Converted value | ULLONG_MAX, errno == ERANGE |
| wcsxfrm() | Length of transformed wide string | >= n |
| wctob() | Converted character | EOF |
| wctomb(), s != NULL | Number of bytes stored | -1 |
| wctomb_s(), s != NULL | Number of bytes stored | -1 |

# ۱۲ ابزارهای مورداستفاده

IDE: Microsoft Visual Studio 2013 Update 5

Compiler: Intel C++ Compiler 15 (Intel Parallel Studio XE 2015 Update 2 Cluster Edition), Microsoft Visual C++ Compiler

Debugger: Microsoft Visual Studio 2013 Update 5 Built-in Win32 Debugger

Disassembler: Microsoft Visual Studio 2013 Update 5 Built-in Disassembler

مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

# ۱۳ منابع


1- Secure Coding in C and C++ Second Edition

نویسنده: Robert C. Seacord

انتشارات: Addison-Wesley Professional

ISBN-10: 0-321-82213-7

ISBN-13: 978-0-321-82213-0

2- Secure Programming Cookbook for C and C++

نویسنده: John Viega

انتشارات: O'Reilly Media

ISBN-10: 0596003943

ISBN-13: 978-0596003944

3- Cert.Org Secure Coding

https://securecoding.cert.org

4- SEI Digital Library

http://resources.sei.cmu.edu


مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای