

Forum:[Hack Forums](#)

Thread: [Pointers](#)

spl0itpid1 - May 27, 2022, 9:26 am

Aloha

When I will implement new code for my own it start be hard to get idea how to use Pointers correctly. Especially pointers to pointers, use them in functions etc. With short code it isn't problem but when make little complicated, there confusing arrived.

Do you have some practical resources to master that? I would like to make upper my skill in that field.

```
Code void addNode(Node **root, int data) {
    if(*root == NULL) {
        *root = createNode(data);
        if(*root) {
            (*root)->data=data;
            (*root)->left=(*root)->right=nullptr;
        }
    }
    else {
        if((*root)->data >= data) {
            addNode(&(*root)->left,data);
        }
        else {
            addNode(&(*root)->right,data);
        }
    }
}

int main() {
    Node* root = NULL;
    addNode(&root, 12);
}
```

Lynx - May 27, 2022, 9:29 am

A double pointer, especially in linked lists, is used to change the content of the variable to give the address of. In this case it is "root" which points to the head of the linked list. Let's say you have a function that removes the first node in the list; You need to remove that node but also make sure the variable you have pointing to the head now points to the new head.

spl0itpid1 - May 27, 2022, 10:03 am

(May 27, 2022 09:29 AM)Lynx Wrote: [\[CITE\]](#) A double pointer, especially in linked lists, is used to change the content of the variable to give the address of. In this case it is "root" which points to the head of the linked list. Let's say you have a function that removes the first node in the list; You need to remove that node but also make sure the variable you have pointing to the head now points to the new head.

Ok, so.. create pointers to struct object :

Code Node* root = NULL;

```
addNode(&root, 12);
```

Next we pass it in function, but why with `**root`? why not just use `*root`? and in main functions `addNode(root)`?

```
Code void addNode(Node **root, int data)
```

And This is still confusing me .. `&(*root)` what exactly that mean and why that is used?

```
Code addNode(&(*root)->right,data);
```

In what case is mostly used pointers, include malware?

realsunjester - May 27, 2022, 8:10 pm

(May 27, 2022 10:03 AM)spl0itpid1 Wrote: [\[CITE\]](#) Ok, so.. create pointers to struct object :

```
Code Node* root = NULL;
```

```
addNode(&root, 12);
```

Next we pass it in function, but why with `root`? why not just use `root`? and in main functions `addNode(root)`?

```
Code void addNode(Node **root, int data)
```

And This is still confusing me .. `&(*root)` what exactly that mean and why that is used?

```
Code addNode(&(*root)->right,data);
```

In what case is mostly used pointers, include malware?

is a pointer to a location in memory

is a pointer to a pointer in memory

Sajuuk - May 28, 2022, 12:22 am

(May 27, 2022 10:03 AM)spl0itpid1 Wrote: [\[CITE\]](#) And This is still confusing me .. `&(*root)` what exactly that mean and why that is used?

```
Code addNode(&(*root)->right,data);
```

It's passing the first argument of the `addNode()` function as a variable. The `"->"` always throws me off, since I'm used to doing things the long way when it comes to pointers. (Simplify it for myself. I can read code better when each step is laid out right there.) It's pointing to the variable's data.

Pointer is to the address of (a.k.a. the ampersand) the pointer to `"root"`, and finally the value of the `"right"` variable led to by said pointer. I assume the comma is to separate arguments, so `"data"` is passed as the second variable. Whatever it's defined as within the program itself without the use of pointers from what the code shows, so nothing complex to be explained pointer-wise for the second parameter.

<https://www.geeksforgeeks.org/arrow-operator-in-c-c-with-examples/>

(May 27, 2022 10:03 AM)spl0itpid1 Wrote: [\[CITE\]](#) In what case is mostly used pointers, include malware?

Well, the idea is to avoid detection. Pointers are used to shuffle around certain data within memory to mask its purpose. Rather than loading an entire exploit into memory, you can shuffle around pieces of its code as a sort of workaround when it comes to the AV's active monitoring, then reassemble it after at least one pass through memory. Modern AVs constantly scan memory. Causes a huge CPU load, and makes it way harder to load and execute exploits from memory.

spl0itpid1 - May 30, 2022, 12:16 pm

Thx, I spent few big hours to train pointers with your advice and more, and now I can say that understand clearly upper code and more ;>

Little confusion get me with diffrents with `*root->value` and `(*root)->value` ... `(*root)->value` we use when in function we call `**root`?

```
Code void printList(LIFO **root) {  
    if(*root){  
        std::cout<<(*root)->value;
```

```

*root=(*root)->next;

    printList(root);
}
else
    return;

```

for example this function run without error, but when try *root->value than error arrived. Why and what are diffrents in that?

(May 28, 2022 12:22 AM)Sajuuk Wrote: [\[CITE\]](#) the idea is to avoid detection. Pointers are used to shuffle around certain data within memory to mask its purpose. Rather than loading an entire exploit into memory, you can shuffle around pieces of its code as a sort of workaround when it comes to the AV's active monitoring, then reassemble it after at least one pass through memory. Modern AVs constantly scan memory. Causes a huge CPU load, and makes it way harder to load and execute exploits from memory.

Thanks! Do you recommend any public code with usage pointers in malware to study? For that I mean some piece of code that will readable for newbie in this topic :>

Sajuuk - May 31, 2022, 8:31 pm

(May 30, 2022 12:16 PM)spl0itpid1 Wrote: [\[CITE\]](#) Thx, I spent few big hours to train pointers with your advice and more, and now I can say that understand clearly upper code and more ;>

Little confiusion get me with diffrents with *root->value and (*root)->value ... (*root)->value we use when in function we call **root?

```

Code void printList(LIFO **root) {
    if(*root){
        std::cout<<(*root)->value;
        *root=(*root)->next;

        printList(root);
    }
    else
        return;
}

```

for example this function run without error, but when try *root->value than error arrived. Why and what are diffrents in that?

Thanks! Do you recommend any public code with usage pointers in malware to study? For that I mean some piece of code that will readable for newbie in this topic :>

The double asterisk/star is more pointer bullshit which I'm not terribly familiar with. It's all about direction when it comes to pointers. I do know, off the top of my head, a double star is a "pointer to a pointer", what I call a "double pointer", so:

<https://www.tutorialspoint.com/double-pointer-pointer-to-pointer-in-c>

In the example shown on that page, they initialize 'p2' as a double pointer, then assign its value as the address of 'p1'. The value of 'p2' in the example I linked is the address of the pointer to 'p1'. From what I've picked up on how C/C++ pointers work in the last decade, passing 'p2' by value will pass the address of 'p1', but passing the pointer will in-turn pass the value of 'p1' since 'p2' is a pointer to a pointer. I think that works because of something in the compiler systems, like garbage collection or whatever. (Bear in mind, I've been a chronic mad drunk for months now.) I don't know

exactly how that value comes from nothing more than a pointer-to-a-pointer, but that's as much as I can explain it as. I'm very much simple-minded, and that's the primary reason I've not made any professional career from the wealth of useless programming knowledge I hold.

If you ask me, in terms of pointer logic, the "pointer to pointer" deal is needlessly complex. Granted, I cannot propose a better solution for representing the value of a variable assigned as the address of a pointer to the value of some other variable... but that's likely because I'm some simple-minded hobbyist programmer. :/

Here's some other links relative to double pointers:

<https://www.geeksforgeeks.org/double-pointer-pointer-pointer-c/>

<https://stackoverflow.com/q/5580761>

<https://youtu.be/TPNYntM-5o4> (skip to around 05:45)

EDIT: That Jenny woman (video linked in the line above) has covered a ton of niche topics when it comes programming. Just after a quick skim over, she's got a lot of videos in which she goes quite in-depth in relation to whatever's being discussed. Woman seems to know what she's talking about. Maybe she can better explain how double pointers work. I noted the timestamp of where she crosses over into double pointers (from single pointers).

Faith Connors - June 1, 2022, 6:14 pm

hmm, there are enough good explanations here that i think you can probably grasp it fairly well now. so instead, i'll offer an actual use-case relevant to what you've asked for (malware-related).

when antiviruses do static analysis on a file, one of the things they look for is what functions are actually linked and called by the binary. they're looking for functions that are commonly used for certain tasks in malware, like opening a process [`OpenProcess()`] and reading/writing its memory [`WriteProcessMemory()`]. despite not being inherently malicious and having enough plausible use-cases, these are still functions that malware devs will frequently use in their code, and as a result, antiviruses will look for these functions and figure out how they are used in a binary to determine if it's malicious.

so what if you could access these functions without actually writing these functions in your code?

in other words, instead of just importing `<windows.h>` and calling the functions like you normally would, what about dynamically loading the .dll that contains them, and writing the `__stdcall` function-signature so that you could call it, without actually telling the compiler to link the function?

better yet, what if the dll's were already loaded and all you needed to do was get a pointer to the function already in memory?

introducing, the process environment block (PEB). featuring, the `PEB_LDR_DATA` structure:

and more importantly, both the `LIST_ENTRY` and `LDR_DATA_TABLE_ENTRY` structs:

the PEB contains all sorts of info about the actual process running that is important to the kernel. for our purposes, we can use it to find the offsets of dll's that are loaded by the process.

and windows loads all its important win32 dll's for every process (ntdll, kernelbase, kernel32, etc.) at that, it also loads them to the same memory addresses for every process, every time a new process is spawned.

(so if you know where they are loaded for your process, you also know where they are loaded for all other processes.)

now, to reiterate, we need to find the addresses of dll's loaded in the system.
we need to find the DllBase field from the LDR_DATA_TABLE_ENTRY struct.

so, when we read PEB_LDR_DATA, we have a LIST_ENTRY struct, which you might recognize as a doubly-linked list (it's circular too.)

*Flink is the pointer to the next LIST_ENTRY forwards, and *Blink is a pointer to the LIST_ENTRY behind it.

each LIST_ENTRY struct points to the LIST_ENTRY struct in each LDR_DATA_TABLE_ENTRY.

so we will also need to get the offset to the base of the LDR_DATA_TABLE_ENTRY. luckily, winapi has the CONTAINING_RECORD macro to help us with that...

no full code samples. if you want a version to compile out-of-the-box, DM me i guess.

Code PPEB_LDR_DATA pLdr = pPeb->Ldr;

```
PLIST_ENTRY startPos = &pLdr->InMemoryOrderModuleList;
```

```
PLIST_ENTRY currPos = startPos->Flink;
```

```
int moduleCount = 0;
```

```
while (currPos != startPos) {
```

```
    moduleCount++;
```

```
    currPos = currPos->Flink;
```

```
}
```

```
printf("Found %d modules loaded in memory.\nPrinting their details...\n\n", moduleCount);
```

```
currPos = currPos->Flink;
```

```
PLDR_DATA_TABLE_ENTRY pLdrEntry;
```

```
for (int i = 0; i < moduleCount; ++i) {
```

```
    pLdrEntry = CONTAINING_RECORD(
```

```
        currPos,
```

```
        LDR_DATA_TABLE_ENTRY,
```

```
        InMemoryOrderLinks);
```

```
    printf(
```

```
        "Path: %ls, Base Address @ 0x%p, Entry Point @ 0x%p\n",
```

```
        pLdrEntry->FullDllName.Buffer,
```

```
        pLdrEntry->DllBase,
```

```
        pLdrEntry->Reserved3[0]
```

```
    );
```

```
    currPos = currPos->Flink;
```

```
}
```

essentially, the -> operator is just a shorthand notation for dereferencing a pointer and accessing part of its data.

Code #include <stdio.h>

```
struct car {
    char color[10];
    int mileage;
};

int main() {
    car x = car{ "red", 5 };
    car* p = &x;
    printf("%p\n", p);
    printf("%s\n", p->color); //this is the important part
    printf("%s\n", (*p).color); //equivalent to this
    return 0;
}
```

i will also note that the above explanation of PEB and LDR is overly-complicated, and no malware does this verbatim, i've just done it for visual purposes, BUT there is malware out there which does access the PEB for other data, like determining if there's a debugger attached to the process (smoke loader)

(hint: the PEB is always located at fs:[30h])

SIM-3364521 - June 2, 2022, 10:19 am

I don't think there is a secret to mastering pointers. The concept is very simple though.

There is a chance you just didn't get a good introduction into the topic yet.

- Make sure you understand how memory works.
- Pointers are also just variable types like any other, only that they hold an address to another memory location

From there i really think that it is just practice. You will surely get the hang of them after a short while. After about one month coding in C or C++ every day for a short while pointers should become second nature to you.

Good Luck on your journey there!

spl0itpid1 - June 2, 2022, 1:26 pm

Yay!!! I'm very excited that you have replied so much valuable responds. I'm sure it will help not only me but every user that have problems with pointers too.

I did today all "have to" jobs so I have all night for study deeply your replies! :>

(Jun 1, 2022 06:14 PM)Faith Connors Wrote: [\[CITE\]](#) no full code samples. if you want a version to compile out-of-the-box, DM me i guess.

?

Mr Roboto - June 3, 2022, 4:17 am

(Jun 1, 2022 06:14 PM)Faith Connors Wrote: [\[CITE\]](#) hmm, there are enough good explanations here that i think you can probably grasp it fairly well now.

so instead, i'll offer an actual use-case relevant to what you've asked for (malware-related).

when antiviruses do static analysis on a file, one of the things they look for is what functions are actually linked and called by the binary. they're looking for functions that are commonly used for certain tasks in malware, like opening a process [`OpenProcess()`] and reading/writing its memory [`WriteProcessMemory()`]. despite not being inherently malicious and having enough plausible use-cases, these are still functions that malware devs will frequently use in their code, and as a result, antiviruses will look for these functions and figure out how they are used in a binary to determine if it's malicious.

so what if you could access these functions without actually writing these functions in your code?

in other words, instead of just importing `<windows.h>` and calling the functions like you normally would, what about dynamically loading the .dll that contains them, and writing the `__stdcall` function-signature so that you could call it, without actually telling the compiler to link the function?

better yet, what if the dll's were already loaded and all you needed to do was get a pointer to the function already in memory?

introducing, the process environment block (PEB). featuring, the `PEB_LDR_DATA` structure:

and more importantly, both the `LIST_ENTRY` and `LDR_DATA_TABLE_ENTRY` structs:

the PEB contains all sorts of info about the actual process running that is important to the kernel. for our purposes, we can use it to find the offsets of dll's that are loaded by the process.

and windows loads all its important win32 dll's for every process (ntdll, kernelbase, kernel32, etc.) at that, it also loads them to the same memory addresses for every process, every time a new process is spawned.

(so if you know where they are loaded for your process, you also know where they are loaded for all other processes.)

now, to reiterate, we need to find the addresses of dll's loaded in the system.
we need to find the `DllBase` field from the `LDR_DATA_TABLE_ENTRY` struct.

so, when we read `PEB_LDR_DATA`, we have a `LIST_ENTRY` struct, which you might recognize as a doubly-linked list (it's circular too.)

*`Flink` is the pointer to the next `LIST_ENTRY` forwards, and *`Blink` is a pointer to the `LIST_ENTRY` behind it.

each `LIST_ENTRY` struct points to the `LIST_ENTRY` struct in each `LDR_DATA_TABLE_ENTRY`.

so we will also need to get the offset to the base of the `LDR_DATA_TABLE_ENTRY`. luckily, `winapi` has the `CONTAINING_RECORD` macro to help us with that...

no full code samples. if you want a version to compile out-of-the-box, DM me i guess.

```
Code    PPEB_LDR_DATA pLdr = pPeb->Ldr;
```

```
PLIST_ENTRY startPos = &pLdr->InMemoryOrderModuleList;
PLIST_ENTRY currPos = startPos->Flink;
int moduleCount = 0;
```

```

while (currPos != startPos) {
    moduleCount++;
    currPos = currPos->Flink;
}

printf("Found %d modules loaded in memory.\nPrinting their details...\n\n", moduleCount);

currPos = currPos->Flink;
PLDR_DATA_TABLE_ENTRY pLdrEntry;
for (int i = 0; i < moduleCount; ++i) {
    pLdrEntry = CONTAINING_RECORD(
        currPos,
        LDR_DATA_TABLE_ENTRY,
        InMemoryOrderLinks);
    printf(
        "Path: %ls, Base Address @ 0x%p, Entry Point @ 0x%p\n",
        pLdrEntry->FullDllName.Buffer,
        pLdrEntry->DllBase,
        pLdrEntry->Reserved3[0]
    );

    currPos = currPos->Flink;
}

```

essentially, the -> operator is just a shorthand notation for dereferencing a pointer and accessing part of its data.

Code #include <stdio.h>

```

struct car {
    char color[10];
    int mileage;
};

int main() {
    car x = car{ "red", 5 };
    car* p = &x;
    printf("%p\n", p);
    printf("%s\n", p->color); //this is the important part
    printf("%s\n", (*p).color); //equivalent to this
    return 0;
}

```

i will also note that the above explanation of PEB and LDR is overly-complicated, and no malware does this verbatim, i've just done it for visual purposes, BUT there is malware out there which does access the PEB for other data, like determining if there's a debugger attached to the process (smoke loader)

(hint: the PEB is always located at fs:[30h])
 Very informative post. Thanks!