

Linux - System Call Inherit

Group 12

Prankur Gupta

Sumit Bagga

Abhishek shukla

108492684

108235636

107598884

{prgupta, ssbagga, ashuklaravis}@cs.stonybrook.edu

April 29,2012

1 Abstract

System calls provide the interface between a process and the operating system (kernel). We as a user process, can request kernel to perform some privileged tasks, by calling these system calls. When a user process invokes a system call, the CPU switches to Kernel Mode and starts the execution of kernel function. There is a set of such system calls defined in Linux, represented by a unique system call number. For each of these system call, there is a unique system call handler. Any changes in the system call or their handler, must be followed by recompiling the kernel so that the changes can take affect. So, it is neither easy to change or add a system call as per users need, nor does Linux provides some implementation for doing that. Adding a new system call to the kernel and then creating programs that use this call take away the program's portability. Adding a system call can also introduce a serious security problem into your system. The best solution is to write our own system calls but adding them statically or even as a module requires changes in the kernel code which may cause inconsistency and takes away kernel portability. Unlike Linux, BSD provides facilities to override the default system call vector. We aim to provide user an additional functionality to override the system call vector and also its subsequent child processes will also use that overridden vector only. For this we created a module to register and deregister the vector names, which have the overridden system call number and their function implementation. The user can choose to use the overridden system call vector through `ioctl`, which populates the private data field of `task_struct` of that process with the address of the list of system call and their function implementation.

2 Introduction

In Linux, accessing the *sys_call_table* by user process or even LKM (Loadable Kernel Module) is not possible. And in order to add a new system call we need to add an entry in the *syscall.h*. But there can be a need of adding new functionality in form of system calls. So, we need an implementation for providing that functionality. We came across many ideas for implementing this e.g.

- (a) creating a new system call for this purpose,
- (b) intercepting system calls in user level library,
- (c) overriding the global system call table.

Among these three, we decided to go with the last one, as it seemed reasonably better in accordance to what we aim to achieve. It is sometimes required that an application wants its processes to use a set of system calls only and might want their own implementation of those system calls, e.g. An application while calling `exec()` might want to log this information in a file, so we can wrap their `exec()` syscall to do this task.

Now, if we want to override the global system call table then we need to identify which process needs what additional functionality. We can't just create another syscall table and ask our process to use that if they needed additional functionality. We needed a way to generalise it. So, we decided to create new overridden system call tables on per process basis. As this provides user specific functionalities.

We are maintaining a per-process system call vectors information in the private field of their *task_struct*. The process needs to choose the system call vector beforehand i.e. before calling any system call, through `ioctl` of the *"ioctl_device"* character device loaded also via module. The list of all the possible overridden system call vectors can be available in *"/proc/"* filesystem. The reason for choosing *"/proc/"* filesystem is because its easy to understand and handling kernel internal data structures inside it is also easy. A user process can override the default *sys_call_table* with any of these registered system call vectors. Once a user process *task_struct* is populated by the address of the *syscall_vector's* address, whenever this process calls a system call, we access the overridden or wrapped functionality as desired by user. Now when a process is forked or cloned, the *task_struct* is copied to its child, so the child will perform as a shadow of its parent whenever it calls the overridden system call vectors.

3 Background

Each system call has a unique number by which it is represented and a unique handler (internal kernel data structure). This unique number is what is stored in the *eax* register before control is given to kernel code. Each system call has to follow two fixed instructions i.e. *sysenter* and *sysexit*. They represent the entry and exit of the kernel control of the system call. Whenever a user process requests for privileged access through system call, an interrupt is generated **int 0x80**. After this interrupt, context switch happens and all the interrupts are disabled. The interrupt calls the respective system call definition through *sys_call_table* based on the value of the *eax* register. When the desired work is done, the control passes to user space after executing *sysexit* and the interrupts are enabled again.

4 Motivation

System Call Inheritance can prove to be a powerful scheme and can have the following applications:

- a) System calls can be modified to suit application specific usage of the OS. For Ex Logging, Profiling etc
- b) Security Applications: Depending on the security level of the user, the set of system calls allowed can be different.
- c) Additional functionality can be added to the existing system calls fairly easily, maintaining the portability of the kernel. Kernel capable of allowing a user to extend system calls without changing the behaviour of other applications on the system is required.
- d) BSD has the feature of overriding system call table at a process level thus allowing access to kernel data structures without having a risk of writing an unsecure or bug prone kernel code.
- e) By this approach a user can extend system calls without changing behaviour of other applications on system.

This can provide portability of kernel and ease of performing desired privileged work, without modifying a lot of kernel code.

5 Major Features

- (a) System call overriding: User defined functions are overridden when normal system calls are made. This paper discusses one of the possible ways to bring this functionality into the linux kernel.

- (b) System call wrapping: Even though it is easy to implement system call wrapping once system call overriding is implemented, it proves to be important; a feature that could be used for logging purposes.
- (c) System call inheritance: The system call vector which is used to override the system calls in the parent will be inherited into the child process.
- (d) Ioctl: We have used ioctl to add the overriding system call vector to the process's task structure. We chose to use ioctl and not a modified version of exec as we can add/remove/change the system call vectors at runtime.
- (e) Helper functions: We have added all the required helper functions to register, unregister system call vectors etc.

6 Design

In this section we are discussing about our approach to achieve our desired aim. We have designed two *primary* modules and two *helper* modules for testing

- a) Module to register and unregister the system call vector
- b) Module for a character device through which we would be sending ioctl's.
- c) Two modules as an implementation of new system call vectors having overridden or wrapped syscall functions.

In the first module and most important **reg_unreg.ko**, we are registering/deregistering the system call vector. These vectors are visible to the user in the **proc** filesystem in the file named as **syscall_vectors**. We create a file in *proc* filesystem using *create_proc_entry()* function and defines *pde* \rightarrow *read_proc* by our own implemented function i.e. *show_vectors*.

Proc filesystem is used because we do not require users to write in the file, or do any other file operations. Further proc filesystem allows an easy hook to data structures defined in kernel. Also provides easy way to access them using callback functions available like *read_proc* and *write_proc*. Here, *write_proc* is not implemented since we don't want user to write anything to our data structures. This module contains four exported functions:

```
int register_syscall(char *vector_name, unsigned long vector_address); (1)
int unregister_syscall(char *vector_name, unsigned long vector_address); (2)
unsigned long get_vector_address(char *vector_name); (3)
int reduce_ref_count(char *vector_name); (4)
```

As the name suggests, they are used for registering, deregistering ,showing the registered vectors and reducing the reference count of the vectors used which are defined in *syscall_vectors* file in *proc* filesystem. We are using *mutex* on *list_lock* to protect accesses of registering / deregistering / showing / reducing reference count of the vector. We are using the following structure of vector :

```
struct new_vector {
    char vector_name[MAX_VECTOR_NAME_LEN];
    unsigned long vector_address;
    int ref_count;
    struct module *vector_module;
    struct new_vector *next;
};
```

where,

struct module * stores the information of the registering module
vector_address contains the address of the syscall_vector structure

```
struct syscall_vector {
    struct overridden_syscall sys_call;
    struct syscall_vector *next;
};
```

```
struct overridden_syscall {
    int syscall_no;
    unsigned long function_ptr;
};
```

The second module **proc_test.ko** is for registering the character device **ioctl_device** with Device number 121. This device is used for sending two ioctl's namely IOCTL_SET_VECTOR and IOCTL_REMOVE.

IOCTL_SET_VECTOR when called, increases the reference count of this module and retrieves the vector_address of the desired vector name through externed method (3) which in turn increases the reference count of that vector and adds that address into the new **void *** field created inside the **task_struct** of the calling process. The added field in the task_struct is void *syscall_inherit_data.

IOCTL_REMOVE when called, removes the particular vector name from the task_struct of the calling process and also reduces the reference count of the vector name and also calls *module_put()*, thus decreasing the reference count of the module itself.

All these functions of adding/ removing the vector address in the *task_struct* of the calling process are safeguarded by a *mutex* lock.

The module **link_vector.ko** and *file_ops_vector*, create a system call vector i.e. *syscall_vector* structure using a set of system call numbers and their function implementation address, which we want to override. It then registers the vector by calling the exported function (1). Similarly, it can unregister the vector, by calling (2).

Apart from this, we have created a call to our defined function inside system call entry code in *entry_32.S*, which checks whether the *void** field of task struct is NULL or not.

7 Implementation

We now give the implementation details of our design. We used Linux 3.2.2+ kernel for our implementation. For every system call, the kernel jumps to the appropriate handler in the *sys_call_table*, based on the system call number. The system call number is stored in the *eax* register. The system call table is in the form of linked list. We search for system call handler corresponding to the number. The parameters to the system call are passed in the kernel stack and we define handler as *asm linkage* function which informs compiler to take arguments from the stack. After return from the handler, the return value is stored in the *%eax* register.

We check the *void ** field of the *task_struct* of the process, whether it is *NULL* or contains the address to the *syscall_vector*. If it is *NULL*, I'm passing the control as it is, *%eax* register containing a fixed return value, which is then replaced by the value of original *%eax* registered popped from the kernel stack. If it's not *NULL*, we push the passed arguments calculated by *%ebp* into the kernel stack space. If the system call number exists in the overridden vector, we call our implementation of the system call, which uses the arguments directly from kernel stack as it is defined *asm linkage*, and send the return value in *%eax* register, which is in turn returned as the final return value of our function.

7.1 Module for registering vector

In this module, we provide four functions which are exported as explained in design part. The function of this module is to register and de-register the vector name in the */proc* file system. In this file system, we have a new file *syscall_vectors* which contains all the registered vector names. In

this module we also define our own function implementation of *read_proc* present in *proc_dir_entry* so to provide user a facility to *cat* the registered system calls.

The function to add new vector (1), allocates a new *new_vector* structure, populates the fields, with *reference_count* = 0 initially and add the desired vector to the list of registered vectors. This is the function called by any module which wants to add new *system_call_vector/table* to the list of other vectors. The use of *struct module** pointer is to check whether some process is using the vector, and thus that module state is busy. The reference count of module is handled by *try_get_module()* and *module_put()* on that *struct module** pointer.

MUTEX lock has been taken before adding the vector address to the list to ensure mutual exclusion. We want only one module to use this API at a time.

The function to deregister the required vector(2), traverses the list and find the location of desired vector, and removes the vector from the list of vectors only if its *reference_count* == 0. If succeeded, it will remove the vector and deallocate the memory taken by vector. This operation is also protected through *MUTEX*.

The function to retrieve the address of the desired vector (3), is called by *ioctl* module when a process wants to include the vector address to its private data field. If the vector is found in the list of registered vectors, the reference count of the module implementing it, is incremented by calling *try_module_get()* to make sure that the module is not unloaded, and also the reference count of that particular vector goes up by 1. This operation is also protected through *MUTEX*.

The function to reduce the reference count on the vector (4), is also called by *ioctl* module, to reduce the reference count of the vector. This indicates that the process no longer wants to use that overridden system call vector. In this case we will also do a *module_put()* on the module using this vector, which in turn reduces the reference count of that module too.

7.2 Adding system calls in vector

This module implements a particular set of system calls which the user want to override in a fixed vector name. Two example modules *link_vector* and *file_ops_vector* are created. A vector *struct syscall_vector* is a linked list of all the system calls, which the user wants to override, they are defined as *struct overridden_syscall*.

The vector contains all the system calls that this module wants to over-

ride or wrap defined by the *struct overridden_syscall* structure which contains the system call number and the address of its function implementation. Overridden functions are declared as *asm linkage* to tell the compiler that arguments to the function of the new system call are to be taken from kernel stack.

7.3 Communicating via IOCTL

The process in order to override the system call, needs to fill its *task_struct*'s private field (void *syscall_inherit_data) with the address of the structure *syscall_vector*, which it wants to override. So in order to achieve this, we created a character device and our process used the *ioctl()* system call for this device to populate its *task_struct*.

The device created is registered in */proc/devices* with major number 121 (randomly chosen). Before making *ioctl* system call device file needs to be created in */dev* file system. For that *mknod* command is used.

```
syntax : mknod < device_file_path > < device_type > <
          major_number > < minor_number >
```

The device *ioctl_device*, defines two *ioctls* *IOCTL_SET_VECTOR* and *IOCTL_REMOVE*, to set and remove the address of the *syscall_vector* structure from current process' task structure.

IOCTL_SET_VECTOR - This *ioctl* is used by user process to pass the vector name to this module, where this module finds the corresponding address of vector in the list of registered vectors maintained by *reg_unreg* module, and adds that to the *task_struct* of the process. It increases the reference count on itself after setting the *task_struct*.

IOCTL_REMOVE - This *ioctl* is used by user process to clear its private data field, and which in turn also reduces the reference count of the vector name, and the module itself.

In order to maintain consistency of system, both the *IOCTLs* must be called by user process. *IOCTL_SET_VECTOR* marks the beginning of usage of overridden system call function table. *IOCTL_REMOVE* marks ending of its usage.

Here also, we have used *MUTEX* for locking the access to the task structure of the current process.

7.4 Calling new system call handler

In the existing system call handler, the arguments are pushed on to the stack. The *%eax* register will have the system call number. In practice,

entry_32.S calls the syscall table with syscall number in *%eax*. Syscall table uses *%eax* as an index/offset for the table and then calls the respective memory location in which the actual system call is present. System calls are declared as *asmlinkage* forcing the kernel to take the arguments from the kernel stack.

In order to call our function implementation, we first send the control to our function *do_syscall_inherit_check* from *entry_32.S*, where we check whether the private data field of *task_struct* is *NULL* or not. But before calling we push *%eax* into stack to retain the value even after we call our own function which tests if a system call is overridden, as we need the *%eax* value if we find that the function is not overridden, to call the actual system call. [Refer : 1 on 17]

Inside *do_syscall_inherit_check*, we first push all the register values(to maintain their state in case the system call is not overridden) on to the stack and then checks if the desired system call is overridden or not. If the system call is overridden, we call our function implementation by passing the address of the function into *%eax* register and calling it. Otherwise, we return *-9999* into *%eax* register. But before returning, we pop out all the values from the stack which we pushed during the initial phase in our function.

After we return from our call, we compare the return value in the *%eax* register with the top value in the stack (original value *%eax* register which we pushed before our call to *do_syscall_inherit_check* in *entry_32.S*). If found that it is overridden, the call to syscall table is skipped, the *%eax* value that is pushed earlier is popped into a register (it should not be popped into *%eax*, we used *%ebx*) and normal execution is continued. If the system call is not overridden, the earlier pushed *%eax* value is popped into *%eax* and a call to syscall table is executed and the instructions in *entry_32.S* are executed as it would have for a normal system call.

7.5 Arguments Passing

System calls are declared as *asmlinkage* forcing the kernel to take the arguments from the kernel stack. We use this same concept and declare the function implementation of overridden system call as *asmlinkage*. For these overridden functions to work properly, we prepare a stack frame as the kernel would do by pushing the arguments in the right order. We push the original function arguments into the stack by doing some arithmetic operations on the base pointer. [Refer : 2 on 18] We then call our function implementation of the overridden system call, this is done by moving the

address of the function implementation into `%eax` register and then making a call to it. After the function call, we get the return value of the system call function in `%eax` register, we copy it as the return value of our function i.e. `do_syscall_inherit_check`, pop out all the values that we have pushed as arguments to the overridden function, and finally return to `entry_32.S`. So, now `%eax` register is populated with the return value of the function `do_syscall_inherit_check`, which in turn stores the return value of our function implementation of overridden system call.

8 Limitations and Advantages

- (a) We are using `ioctl` as our method of communicating the vector name to kernel so that the `ioctl_device` module can find the `vector_address` corresponding to vector name and add it to the task structure. Along with adding to task structure we are also incrementing the reference count of module implementing the vector asked for by doing a `try_module_get()` on that module.

To reduce the reference count we have to have some mechanism to know when the user process ends, which is not possible unless we monitor state of that process continuously. So instead, we created another `IOCTL` which the user process will explicitly call to notify the kernel to strip the vector address from the task structure of that process. In this call itself we reduce the reference count of the module, implementing the vector, by doing `module_put()` on that module.

Now the drawback in this approach is that the user process has to explicitly do a `remove vector` `ioctl` call so that reference counts are updated. So, if the user process, having new vector address in its task structure, is killed in between, while it is still working, will leave all modules in inconsistent state. (But this scenario will cause modules to be in inconsistent state, if we had used our own defined exec system call, other approach that we mentioned in proposal.)

Also this can't be called a drawback completely, it has a hidden advantage as well. It allows a user process to decide when it wants to use the overridden system calls and when original system calls. So, in a single process execution a process can call `IOCTL_SET_VECTOR` to use one vector at one time say 'file_ops_vector' then call `IOCTL_REMOVE` to remove it and then again call `IOCTL_SET_VECTOR` to use another vector say 'link_vector' and then remove it as well and use third vector

or use original system calls altogether.

This way single process at multiple times can use multiple vectors or original system call depending on user requirements. Also, observe that this is an advantage of using `ioctl` over our own defined `exec` system call, which takes in only one vector at a time and will use that throughout its execution.

- (b) Other limitation is that, since we have made a change in `entry_32.S` to change the flow of execution of system call interrupt to have a check whether the process is using any overridden vector or not, this makes our implementation architecture dependant. But again, this is not much of a drawback, because otherwise also, for normal execution all architectures have different way of handling the system call interrupt. That makes our implementation an adhesive part of normal system call interrupt, and a more reliable and stable way of implementation for future standard implementation of *system_call inheritance*.
- (c) One more drawback of having a new call `do_check_syscall_vector` in `entry_32.S`, which checks for `void* syscall_inherit_data` in task structure if it is empty or not, is that now a lot of CPU cycles are wasted in calling the function and doing the check. If we are inheriting only say 5 system calls, this check being made for all the system calls by any process, consumes lots of memory cycles and affects performance. We here leave a scope of improvement over this approach for future implementations and ideas.

9 Conclusion

With the advent of Linux Kernel, the maintainers have made it more and more difficult to change the flow of system calls, may it be adding new system calls or system call overriding. In order to keep people from doing potential harmful things `sys_call_table` is no longer exported. In this paper we propose a mechanism to define new syscall vectors in which the existing system calls can be overridden or wrapped. We maintain the *asmlinkage* functionality in the kernel by adding statements to maintain the kernel stack and by declaring the overriding/wrapping functions as *asmlinkage*. Since the users usually override/wrap few system calls only, we chose to add the syscall vector as a linked list and not as an array. This, not only helps us in adding other information necessary with the overridden/wrapped system calls but also saves precious kernel memory. We chose to implement an *ioctl*

instead changing the existing `exec`(and alike) or having a new version of `exec`, for maintaining kernel portability. We do this to provide an user the ability to change/remove syscall vectors at runtime. Also, we have implemented the inheritance of this new syscall vector that we have added by making changes to the process *task structure*; by adding a *void ** field at the end of the structure thus making sure that we don't disturb the existing kernel code. We have successfully tested scenarios which we think are apt and these have been mentioned in appendix A.

10 References

- (a) <http://lxr.fsl.cs.sunysb.edu/linux/source/>
- (b) <http://www.informit.com/articles/article.aspx?p=370047>
- (c) <http://www.ibiblio.org/gferg/ldp/GCC-Inline-Assembly-HOWTO.html#s2>
- (d) <http://www.cs.lmu.edu/~ray/notes/x86assembly/>
- (e) http://articles.manugarg.com/systemcallinlinux2_6.html
- (f) <http://www.win.tue.nl/~aeb/linux/lk/lk-4.html>
- (g) <http://linux.die.net/lkmpg/x978.html>
- (h) <http://kernelnewbies.org/FAQ/asmlinkage>
- (i) <http://kernelnewbies.org/FAQ/asmlinkage>
- (j) <http://stackoverflow.com/questions/10060168/is-asmlinkage-required-for-a-c-functi>
- (k) <http://www.win.tue.nl/~aeb/linux/lk/lk-4.html>
- (l) <http://www.csee.umbc.edu/~chang/cs313.s02/stack.shtml>

Appendix

A Test Cases

We plan to have the demo with two syscall vectors that we are adding:

1. `file_ops_vector`

- a. `open(wrapped)` : prints the arguments passed and returns a value indicating that open is only wrapped
- b. `fchown(overridden)`: prints the arguments
- c. `read: (overridden)`: puts data(`random_text`) in the data buffer passed by the user.

2. `link_vector`

- a. `link(overridden)`: prints old pathname and new pathname
- b. `unlink(wrapped)`: prints some statements and returns indicating that this call was just overridden

Test1:

User Program has the new system call vector: *file_ops_vector*.

User Program calls:

- 1. `open` with *test1_file*, *O_CREAT* and 777 as arguments.
- 2. `read` with as arguments 100(random file decriptor), user buffer, 200(read size)
- 3. `fchown` with 11,22,33(all random) as arguments

Expected Output:

- 1. Since open is just wrapped, the arguments are printed on the screen from the wrapped function and original system call is called thus creating the file, *test1_file*
- 2. the overridden read function puts text: *randomtext* into the user buffer.
- 3. Prints the arguments.

Test2:

User program has a new system call vector: *link_vector*.

User Program calls:

- 1. `link` with *link_old* and *link_new* as arguments
- 2. `unlink` with *test1_file* as arguments.

Expected Output:

Since link is just overridden, the arguments are jsut printed.

Since Unlink is wrapped, some statements are printed inside the wrapped function and the wrapped function returns with a value indicating that the

function is just wrapped. Thus, original system call unlink is called with *test1_file* as argument, thus removing that file.

Test3:

User Program has the new system call vector: *file_ops_vector*

We have used fork to create a child process.

Parent Process calls open with test3_parent, O_CREAT and 777 as arguments.

Child Process calls fchown with 11,22,33(all random) as arguments

Expected Output:

Since open is just wrapped, the arguments are printed on the screen from the wrapped function and original system call is called thus creating the file, test3_parent

This test proves to be important as we are showing the functionality of system call inheritance by using fork. This test proves that even the child process' task structure is updated with the parents task struct. Thus, overridden fchown in the child is this called and just prints the arguments.

Test4:

User Program has the new system call vector: link_vector.

We have used fork to create a child process.

Parent Process calls link with test4_oldpath and test4_newpath as arguments

Child Process calls unlink with test3_parent (which was created in test3) as argument.

After child process returns, parent process sleeps for sometime.

Then the parent process removes link_vector from its task structure.

Expected Output: While the parent process is sleeping, we see that link_vector is still associated with it. If we try to remove the vector module, if the reference counts had been handled properly, we should not be able to remove the link vector module. We check this while the process is sleep. After it returns from sleep, the process itself will remove the vector from its task struct. Then after the program exits, you can remove the link vector module successfully.

Test5:

Initially the user Program has the new system call vector: file_ops_vector. Open system call is called with test5_file, O_CREAT, 777 as arguments. Then user program at run time removes this vector, sleeps for sometime and adds the link_vector. Now unlink system call is made with test5_file as arguments. Then user program removes link_vector and then sleeps for sometime and then calls open with test5_new_file, O_CREAT, 777 as arguments.

Expected Output:

test5_file is created because the wrapped open system from file_ops_vector call is called. Then the process removes this vector and adds link_vector to the We added sleep in between so that we can ssh and show that the file is being created while the process sleeps. Then, unlink from the link_vector is called. Then the file test5_file will be removed since the unlink system call is just wrapped in link_vector. We have put a sleep again to confirm this by doing an ls and dmesg. Then link_vector is removed from the process' task structure. Then because of the open system call creates the test5_new_file as expected and through the dmesg command we can confirm that there are no overriding system call vectors.

B Code Changes

- (a) Added a new *void ** field *syscall_inherit_data* in the struct *task_struct* in */usr/src/hw3-cse506g12/include/linux/sched.h*
- (b) Added an exteren declaration of *do_syscall_inherit_check* function in */usr/src/hw3-cse506g12/arch/x86/include/asm/signal.h*
- (c) Modified */usr/src/hw3-cse506g12/arch/x86/kernel/entry_32.S*
- (d) Definition of *do_syscall_inherit_check* function defined in */usr/src/hw3-cse506g12/syscall_inherit/syscall_inherit.c*.
- (e) All the four **LKM**'s (Loadble Kernel Modules) are present in */usr/src/hw3-cse506g12/hw3/os_proj*.

C How To Run

- (a) `git clone ssh://user@scm.cs.stonybrook.edu:130/scm/cse506git-s12/hw3-cse506g12`
- (b) `cd hw3-cse506g12`
- (c) `cp kernel.config .config`
- (d) `make`
- (e) `make modules_install`
- (f) `make install`

- (g) reboot
- (h) `cd /usr/src/hw3-cse506g12/hw3/os_proj`
- (i) `./clean.sh`
- (j) `./make.sh`
- (k) `cd test_demos`
- (l) `./test1 file_ops_vector`
- (m) `./test2 link_vector`
- (n) `./test3 file_ops_vector`
- (o) `./test4 link_vector`
- (p) `./test5`

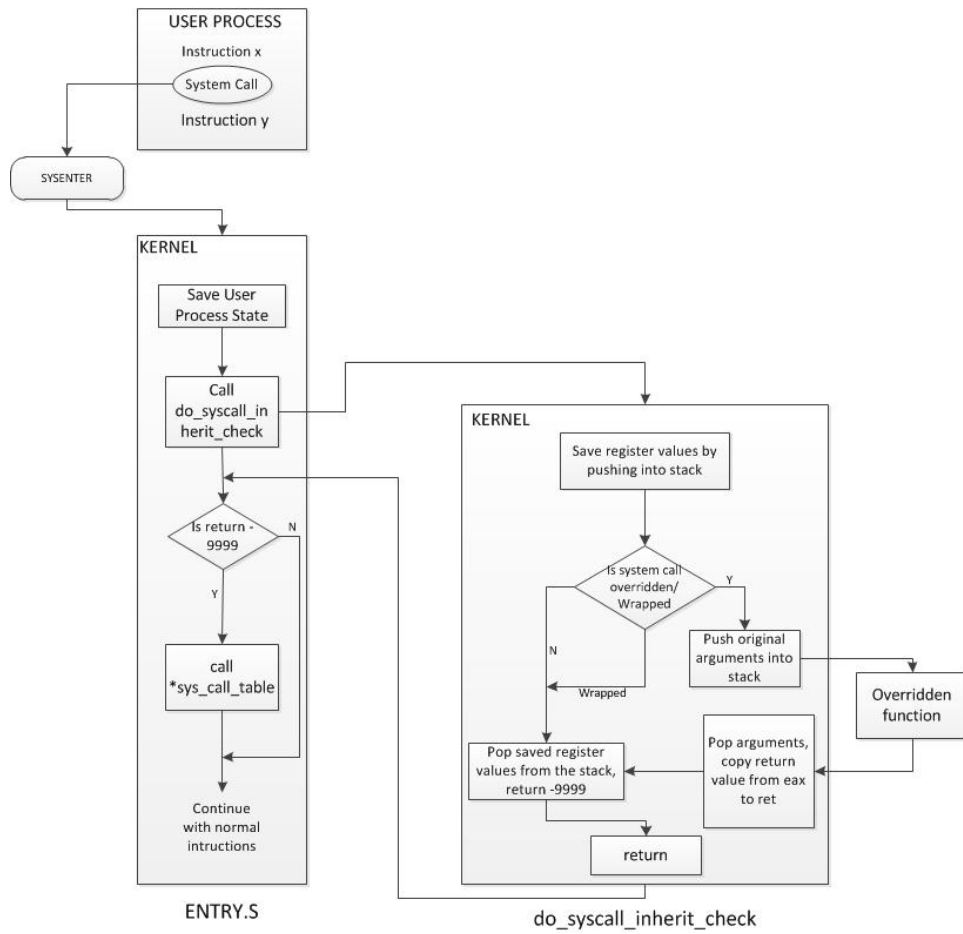


Figure 1: Control Flow

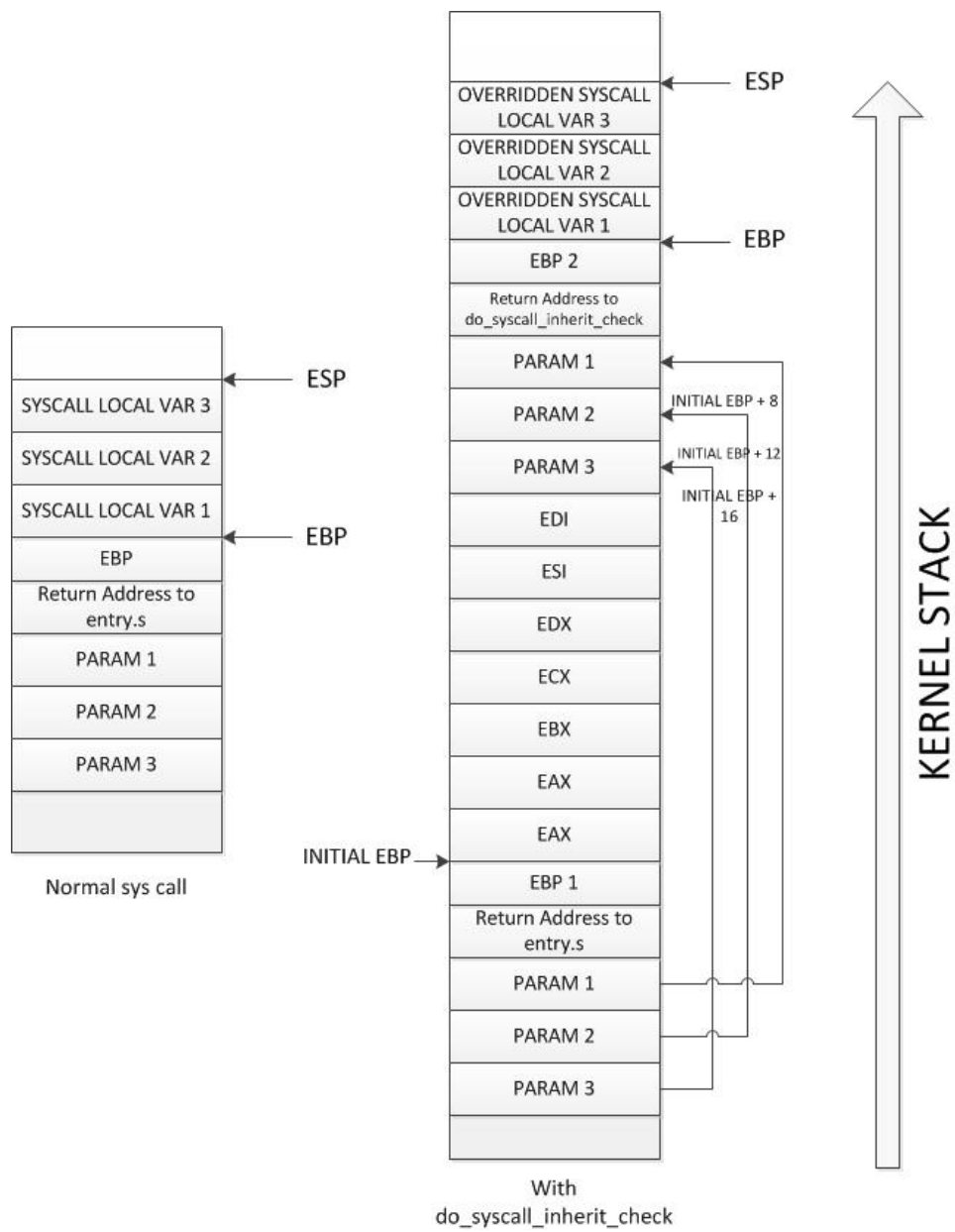


Figure 2: Kernel stack