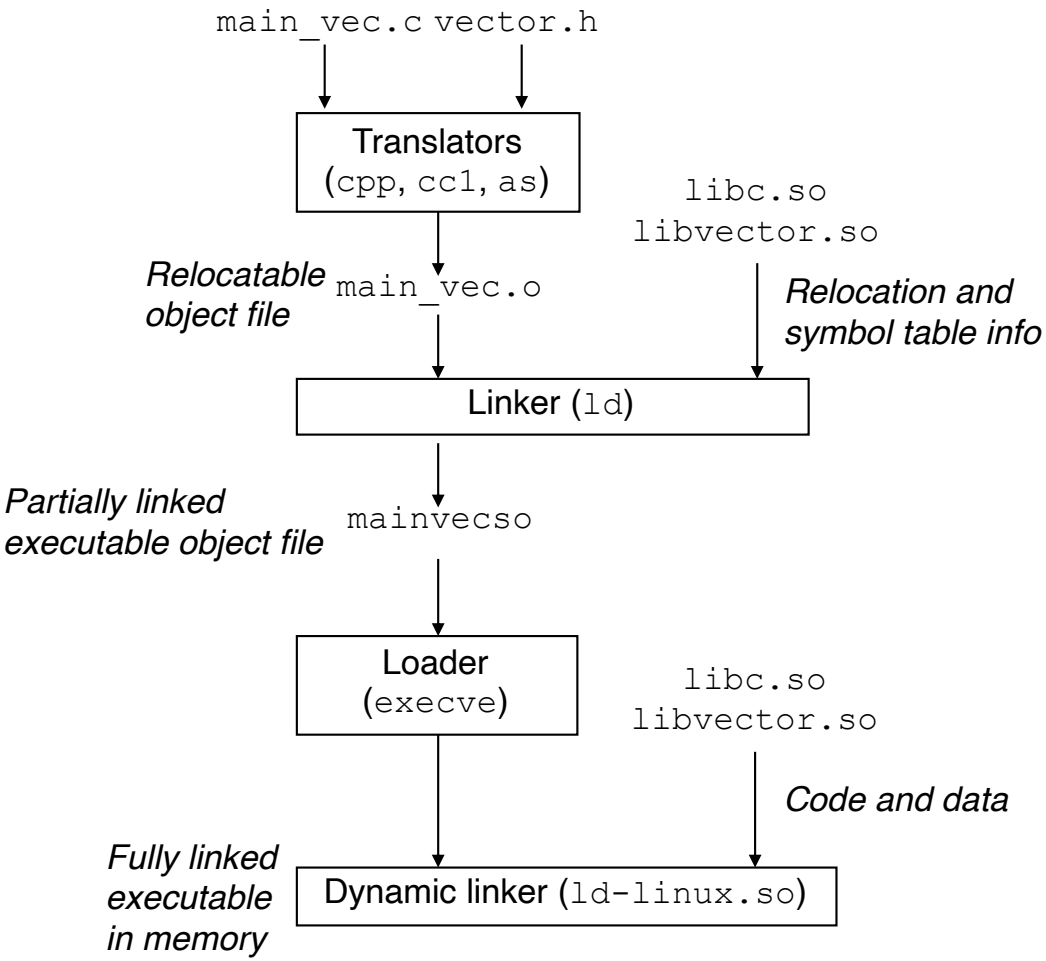
# Dynamic Linking Workflow on Linux



# The Source Files

*vector.h*

```
void addvec(int*, int*, int*, int);
void multvec(int*, int*, int*, int);
int addcount(void);
int multcount(void);
```

*vector_add.c*

```
extern int addcnt;

void addvec(int *x, int *y, int *z, int n) {
  addcnt++;
  for (int i = 0; i < n; i++) z[i] = x[i] + y[i];
}
```

*vector_mult.c*

```
extern int multcnt;

void multvec(int *x, int *y, int *z, int n) {
  multcnt++;
  for (int i = 0; i < n; i++) z[i] = x[i] * y[i];
}
```

*vector_ops_count.c*

```
int addcnt = 0, multcnt = 0;

int addcount(void) {return addcnt;}

int multcount(void) {return multcnt;}
```

*main_vec.c*

```
#include <stdio.h>
#include "vector.h"

int x[2] = {1, 2};
int y[2] = {3, 4};
int z[2];

int main(void) {
  printf("x = [%d %d]\n", x[0], x[1]);
  printf("y = [%d %d]\n", y[0], y[1]);
  addvec(x, y, z, 2);
  printf("z = [%d %d]\n", z[0], z[1]);
  multvec(y, z, x, 2);
  printf("x = [%d %d]\n", x[0], x[1]);
  addvec(z, x, y, 2);
  printf("y = [%d %d]\n", y[0], y[1]);
  return 0;
}
```

*Makefile*

```
VECSRCS := vector_add.c vector_mult.c vector_ops_count.c

libvector.so: ${VECSRCS}
gcc -shared -fpic -o $@ ${VECSRCS}

mainvecso: main_vec.o ./libvector.so
gcc -o $@ main_vec.o ./libvector.so
```

# The Problem

- We have an application binary (`mainvecso`) that links with a dynamically linked library (`libvector.so`), with references that may need run-time resolution.
  - Data references (e.g., to `addcount`, within the library).
  - Code references (e.g., the call to `addvec()` from `mainvecso`).

# The Problem

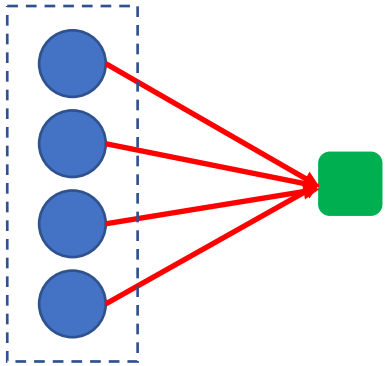
- We have an application binary (`mainvecso`) that links with a dynamically linked library (`libvector.so`), with references that may need run-time resolution.
  - Data references (e.g., to `addcount`, within the library).
  - Code references (e.g., the call to `addvec()` from `mainvecso`).
- In general, the number of references will be much greater than the number of definitions.

# The Problem

- We have an application binary (`mainvecso`) that links with a dynamically linked library (`libvector.so`), with references that may need run-time resolution.
  - Data references (e.g., to `addcount`, within the library).
  - Code references (e.g., the call to `addvec()` from `mainvecso`).
- In general, the number of references will be much greater than the number of definitions.
- How do we design an efficient resolution scheme that will allow us to patch lazily and to share the `.text` section of the library among all of its users?
  - ("Lazy resolution") We want to patch only those references that are truly needed during a program run.
  - ("Once-per-definition resolution") We want to minimize the cost of patching multiple references to a single definition.

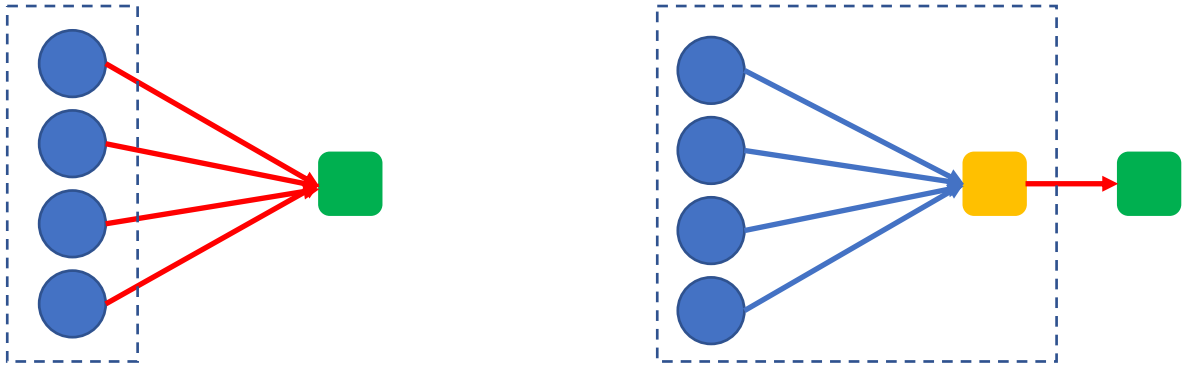
# The Key Insight

- The problem arises because:
  - we have direct connections from multiple sources (references, in module A) to a single destination (definition, in module B); and
  - we have to patch the source end.



# The Key Insight

- The problem arises because:
  - we have direct connections from multiple sources (references, in module A) to a single destination (definition, in module B); and
  - we have to patch the source end.



- If we reconfigure the connections to pass through a common “junction box”, then we can patch the single connection from this box to the definition.
  - The Global Offset Table (GOT) is just this junction box.
  - The Procedure Linkage Table (PLT) is a mechanism to make sure that the blue edges remain PIC.

# The `.got` and `.plt` Sections

- Two special sections in executable ELF files.
  - There may also sometimes be a `.got.plt` section and a `.plt.got` section.
  - For simplicity, ignore this complication.

# The `.got` and `.plt` Sections

- Two special sections in executable ELF files.
  - There may also sometimes be a `.got.plt` section and a `.plt.got` section.
  - For simplicity, ignore this complication.
- The `.got` section is a data section that contains addresses.

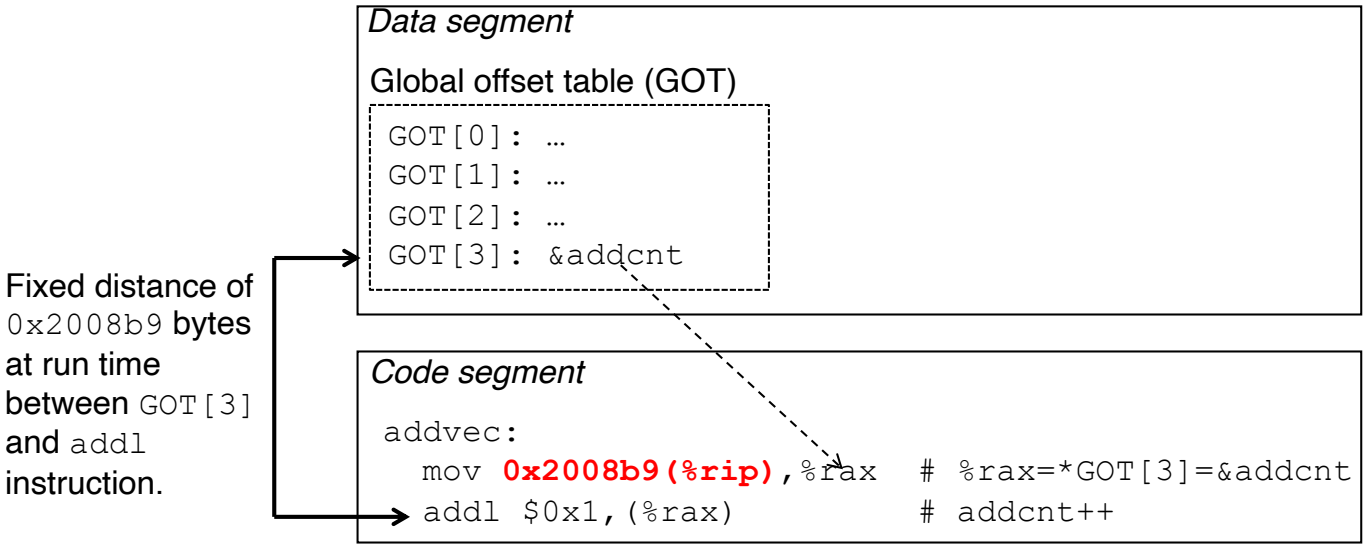
# The `.got` and `.plt` Sections

- Two special sections in executable ELF files.
  - There may also sometimes be a `.got.plt` section and a `.plt.got` section.
  - For simplicity, ignore this complication.
- The `.got` section is a data section that contains addresses.
- The `.plt` section is a code section that contains executable code.
  - Consists entirely of stubs of a well-defined format, dedicated to directing calls from the `.text` section to the appropriate library function.

# The `.got` and `.plt` Sections

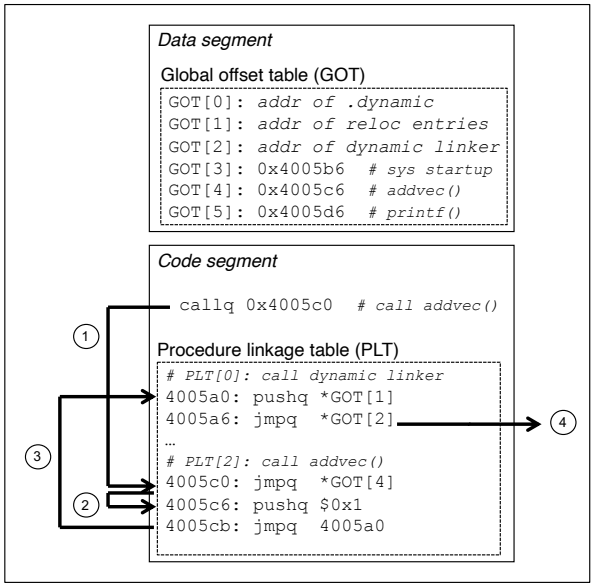
- Two special sections in executable ELF files.
  - There may also sometimes be a `.got.plt` section and a `.plt.got` section.
  - For simplicity, ignore this complication.
- The `.got` section is a data section that contains addresses.
- The `.plt` section is a code section that contains executable code.
  - Consists entirely of stubs of a well-defined format, dedicated to directing calls from the `.text` section to the appropriate library function.
- There will also be `.rel.dyn` and `.rel.plt` sections in the ELF file to enable the dynamic linker to do its job of initializing GOT entries correctly.

# Patching Data References



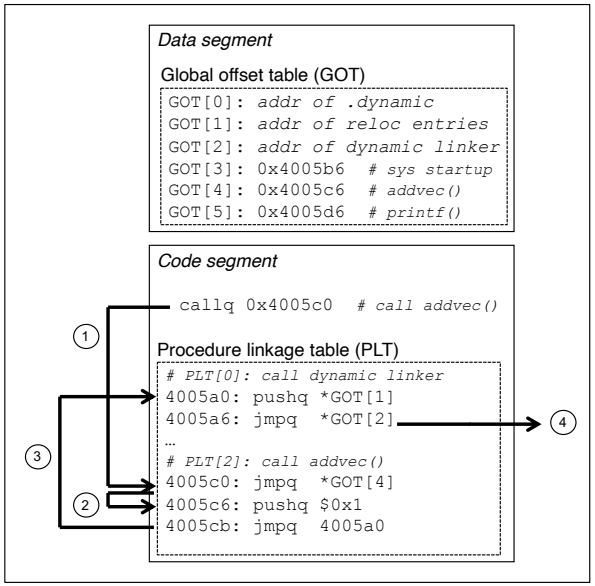

- The dynamic linker relocates each GOT entry to the absolute address of each global variable.
- This uses the special RIP-relative addressing mode, which is available on x86 only in 64-bit mode.
  - A bootstrapping trick is needed in 32-bit mode.

# Patching Code References



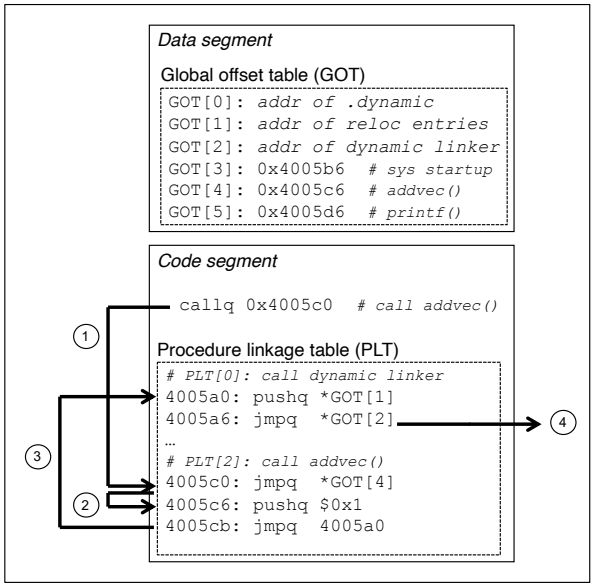

1. The call to `addvec()` reaches the `PLT[2]` stub instead.

# Patching Code References



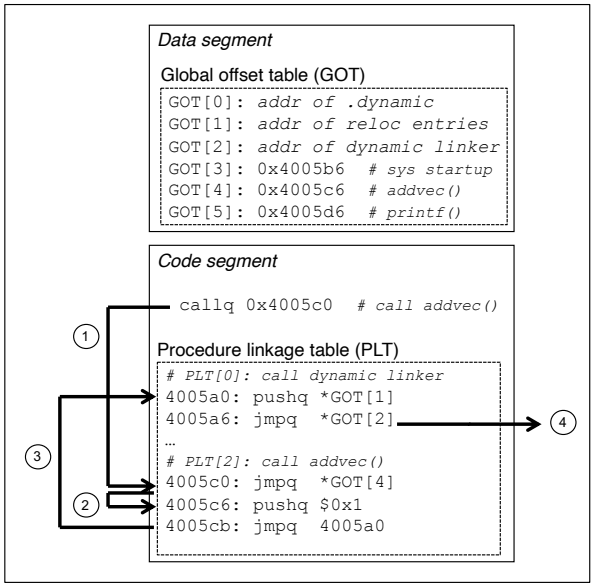

1. The call to `addvec( )` reaches the `PLT[2]` stub instead.
2. The indirect jump through `GOT[4]` reaches the very next instruction (!?).

# Patching Code References



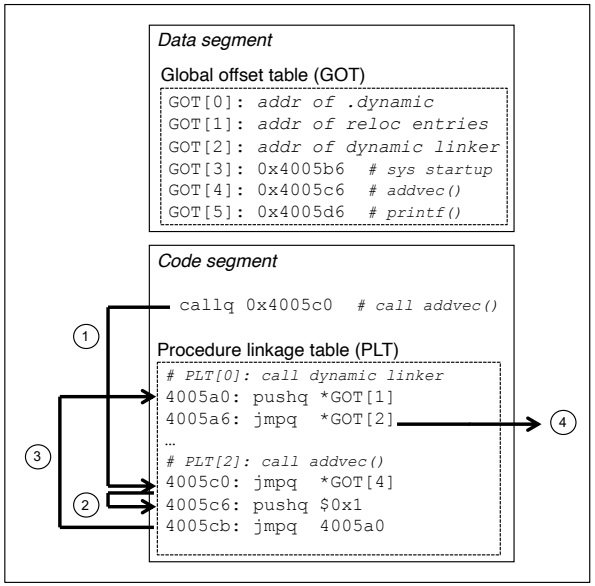

1. The call to `addvec()` reaches the `PLT[2]` stub instead.
2. The indirect jump through `GOT[4]` reaches the very next instruction (!?).
3. After pushing the argument `0x1` on the call stack (uniquely identifying the method), control branches to the `PLT[0]` stub, for the dynamic linker.

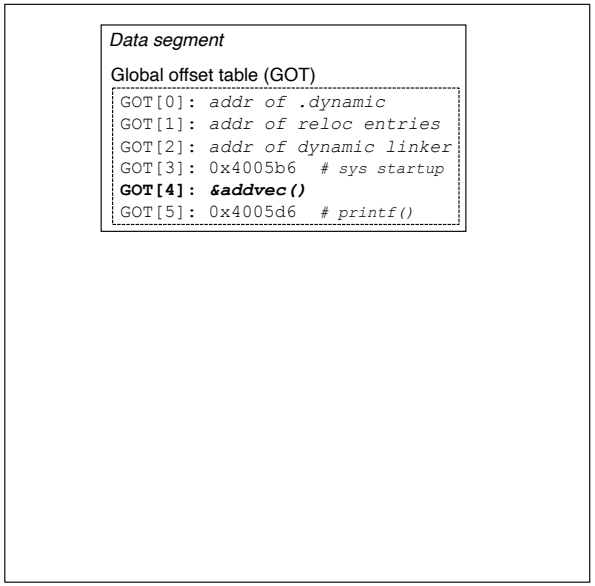
# Patching Code References




1. The call to `addvec( )` reaches the `PLT[ 2 ]` stub instead.
2. The indirect jump through `GOT[ 4 ]` reaches the very next instruction (!?).
3. After pushing the argument `0x1` on the call stack (uniquely identifying the method), control branches to the `PLT[ 0 ]` stub, for the dynamic linker.
4. After pushing another argument on the call stack (to identify the relocation entries), control branches indirectly to the dynamic linker.

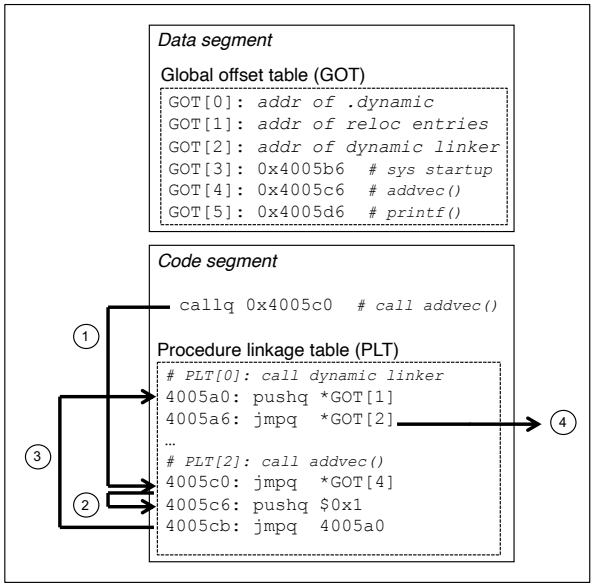
# Patching Code References




1. The call to `addvec()` reaches the `PLT[2]` stub instead.
2. The indirect jump through `GOT[4]` reaches the very next instruction (!?).
3. After pushing the argument `0x1` on the call stack (uniquely identifying the method), control branches to the `PLT[0]` stub, for the dynamic linker.
4. After pushing another argument on the call stack (to identify the relocation entries), control branches indirectly to the dynamic linker.
5. The dynamic linker patches the reference by **updating** `GOT[4]` to the RT address of `addvec()`, and then jumps to that routine.

# Patching Code References





1. The call to `addvec()` reaches the `PLT[2]` stub instead.
2. The indirect jump through `GOT[4]` reaches the very next instruction (!?).
3. After pushing the argument `0x1` on the call stack (uniquely identifying the method), control branches to the `PLT[0]` stub, for the dynamic linker.
4. After pushing another argument on the call stack (to identify the relocation entries), control branches indirectly to the dynamic linker.
5. The dynamic linker patches the reference by **updating** `GOT[4]` to the RT address of `addvec()`, and then jumps to that routine.
6. On subsequent calls to `addvec()`, the indirect jump through `GOT[4]` in step #2 bypasses the following steps and goes directly to the routine.